2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN OPTIMAL STATIC SCHEDULING ALGORITHM
FOR HARD REAL-TIME SYSTEMS
SPECIFIED IN A PROTOTYPING LANGUAGE

by

Julian Jaime Cervantes

December, 1989

Thesis Advisor: Luqi

Approved for public release; distribution is unlimited.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION

1b RESTRICTIVE MARKINGS

2a SECURITY CLASSIFICATION AUTHORITY

2b DECLASSIFICATION / DOWNGRADING SCHEDULE

3 DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution is unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)

5 MONITORING ORGANIZATION REPORT NUMBER(S)

6a NAME OF PERFORMING ORGANIZATION
Naval Postgraduate School

6b OFFICE SYMBOL *(If applicable)*

7a. NAME OF MONITORING ORGANIZATION

6c. ADDRESS *(City, State, and ZIP Code)*
Monterey, CA 93943-5000

7b ADDRESS *(City, State, and ZIP Code)*

8a NAME OF FUNDING / SPONSORING ORGANIZATION

8b OFFICE SYMBOL *(If applicable)*

9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)*

10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

11 TITLE *(Include Security Classification)*
An Optimal Static Scheduling Algorithm for Hard Real-Time Systems Specified in Prototyping Language

12. PERSONAL AUTHOR(S)
Julian Jaime Cervantes

13a TYPE OF REPORT
Master Thesis

13b TIME COVERED
FROM ________ TO ______

14 DATE OF REPORT *(Year, Month, Day)*
1989 December

15 PAGE COUNT
116

16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Computer Aided Prototyping System (CAPS)

19 ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The Computer Aided Prototyping System (CAPS) and the Prototype System Description Language (PSDL) are tools that have been designed to aid in rapid prototyping. Within the framework of CAPS the Execution Support System (ESS) controls the execution of the prototype. The Static Scheduler is the component of the ESS which extracts and realizes critical timing constraints and precedence constraints for operators.

The construction of a Static Scheduling Algorithm provides the foundation for handling hard real-time constraints during the execution of PSDL. The proposed work will be based on the theories of optimal sequencing through modular decomposition, as well as enumeration techniques. An optimal algorithm will provide the analyst with a definitive method for a determining whether a given design can meet its hard real-time requirements.

[illegible] (KR)

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS

21 ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a NAME OF RESPONSIBLE INDIVIDUAL
Lucia Luqi

22b TELEPHONE *(Include Area Code)*
408 646-2735

22c OFFICE SYMBOL
52Lq

Previous editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

S/N 0102-LF-014-6603

An Optimal Static Scheduling Algorithm for Hard Real-Time Systems Specified in a Prototyping Language

by

Julian Jaime Cervantes
Capitao Engenheiro, Força Aérea Brasileira
B.S., Instituto Tecnologico de Aeronáutica, 1978

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1989

Author: ______________________________
Julian Jaime Cervantes

Approved by: Luqi
Luqi, Thesis Advisor

Valdis Berzins
Valdis Berzins, Second Reader

Uno R. Kodres
Uno R. Kodres, Academic Associate
Department of Computer Science

Robert B. McGhee
Robert B. McGhee, Chairman
Department of Computer Science

ABSTRACT

The Computer Aided Prototyping System (CAPS) and the Prototype System Description Language (PSDL) are tools that have been designed to aid in rapid prototyping. Within the framework of CAPS the Execution Support System (ESS) controls the execution of the prototype. The Static Scheduler is the component of the ESS which extracts and realizes critical timing constraints and precedence constraints for operators.

The construction of a Static Scheduling Algorithm provides the foundation for handling hard real-time constraints during the execution of PSDL. The proposed work will be based on the theories of optimal sequencing through modular decomposition, as well as enumeration techniques. An optimal algorithm will provide the analyst with a definitive method for determining whether a given design can meet its hard real-time requirements.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

Many new and sophisticated real-time applications are currently being contemplated by governments and industries around the world. Hard real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. Examples of this type of real-time systems are command and control systems, process control systems, flight control systems, the space shuttle avionics system, future systems such as SDI, and large command and control systems. Most of the hard real-time computer systems are special-purpose and complex, require a high degree of fault tolerance, and are typically embedded in a larger system.[Ref. Luq89 : pp. 417-424]

Also, real-time systems have substantial amounts of knowledge concerning the characteristics of the application and the environment built into the system. A majority of today's systems assume that much of this knowledge is available a priori and, hence, are based on static designs.[Ref. SRC87 : pp. 1-4]

Hard real-time systems are characterized by the fact that severe consequences will result if the timing as well logical correctness properties of the system are not satisfied. Typically a hard real-time system consists of a controlling system and a controlled system, thus the controlled system can be viewed as the environment with which the computer interacts.[Ref. Kic88 : p. 15]

In most of these systems, activities that have to occur in a timely fashion coexist with those that are not time-critical. Let us denote both activities as *tasks* and a task with a timing requirement as a *critical task*. Ideally, the computer should execute critical tasks so that each task will meet its timing requirement, whereas it should execute the non-critical tasks so that the average response time of these tasks is minimized.[Ref. SRC87 : pp. 1-4]

Timing constraints for tasks can be arbitrarily complicated, but the most common timing constraints for tasks are either periodic or sporadic. A sporadic task has a deadline by which it must finish or start, or it may have a constraint on both start and finish times. In the case of a periodic task , period might mean "once per period T". The need to meet the requirements of individual critical tasks is one issue that makes the problem of designing a hard real-time system a difficult problem. In addition to timing constraints, a task is usually subject to other types of constraints such as precedence relationships.[Ref. Kic88 : pp. 80-84]

In summary, hard real-time systems differ from traditional systems in that deadlines or other explicit timing constraints are attached to tasks, the systems are in a position to make compromises, and faults, including timing faults, may cause catastrophic consequences. This implies that, unlike many systems where there is a separation between correctness and performance, in a hard real-time system correctness and performance are very tightly interrelated. Thus hard real-time systems solve the problem of missing deadlines in ways specific to the requirements of the target application.[Ref. LG88 : p. 1]

B. TRADITIONAL SOFTWARE CYCLE AND RAPID PROTOTYPING

1. Traditional Software Cycle

The traditional software cycle and rapid prototyping are two of the more common design methodologies used to maintain a scientific approach to software engineering.

The traditional software cycle is based on the waterfall life cycle, which incorporates individual development stages. Figure 1, below, shows a graphic representation of this approach.

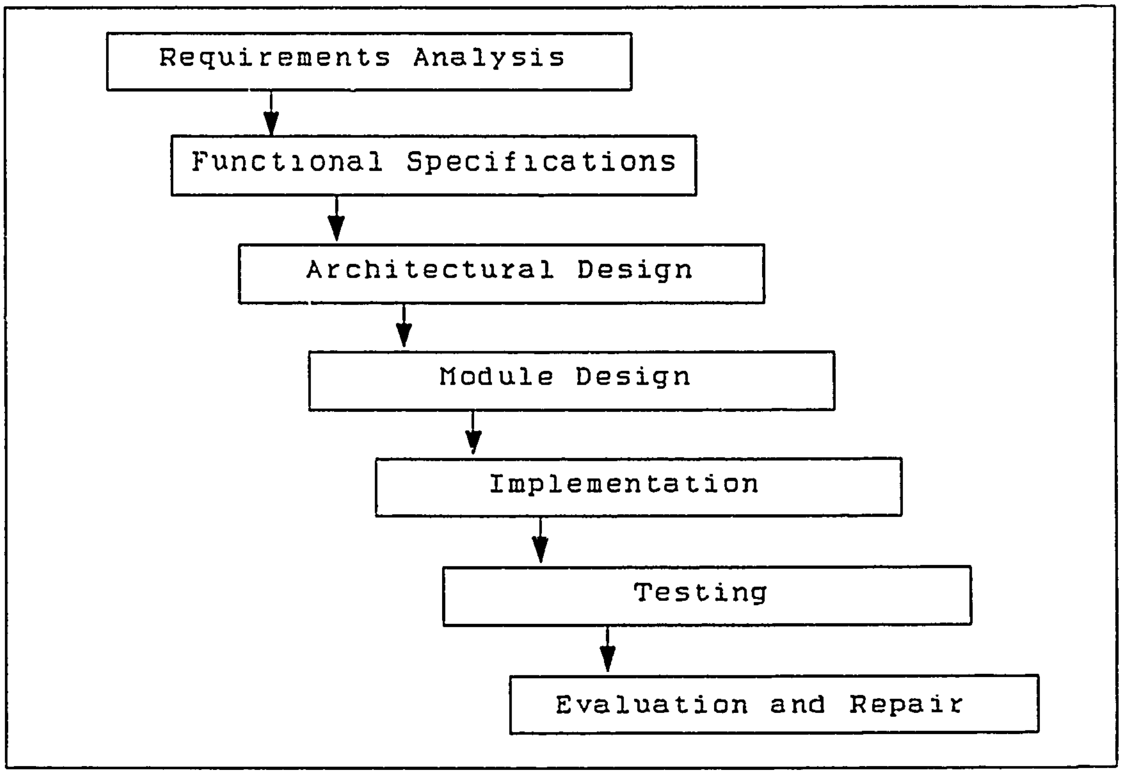


Figure 1 Traditional Software Life Cycle

These stages include requirements analysis, functional specifications, architectural design, module design, implementation, testing,and evolution. Requirements analysis establishes the purpose of the system in development and defines the external interfaces as well the environment within which the system will operate. The functional analysis defines a model of the proposed system, but this model just contains those aspects of the system that are visible to the user. The architectural design generates a high-level model of the system, during this phase the system is partitioned into modules, each of these modules try to hide one specific function, state machine or abstract data type. During the module design phase the algorithm and data structures of each module are defined, in order to realize the behavior specified in the architectural design. The implementation stage is just the coding, in some programming language, of the decisions made durin, ın the module design phase. Testing is the phase when inconsistencies with expected ,erformance are detected. The evolution (or repair) stage is when new features or capabilities are added onto the system in order to meet the requirements of the user, or to repair faults. Depending of the full impact of these faults the overall reliability and accuracy of the system could be in question. The traditional software life cycle yields an executable system only after too much time and money are spent.[Ref. Luq88 : pp. 1-8]

2. Rapid Prototyping

The rapid prototyping methodology is an alternative for the traditional software life cycle, which is proving to be more efficient in design of large hard real-time systems. The goal of rapid prototyping is to develop an executable model of the intended system early in the development process. In general, the prototype is only a partial representation of the intended system and includes only the system's most critical aspects. The code of a prototype usually cannot be used as the final implementation because it may not realize all the aspects of the intended system. Figure 2, on page 5, graphically describes this methodology as a typical feedback loop.

Rapid prototyping initially establishes an interactive process between the user and the designer to concurrently define specifications and requirements for the critical aspects of the system under development. The prototype must satisfy its requirements, and be easy to read and analyze. During demonstrations of the prototype, the user validates the prototype's actual performance. This process continues until the user determines that the prototype meets the time critical aspects of the system under development.[Ref. LK88 : pp. 66-72]

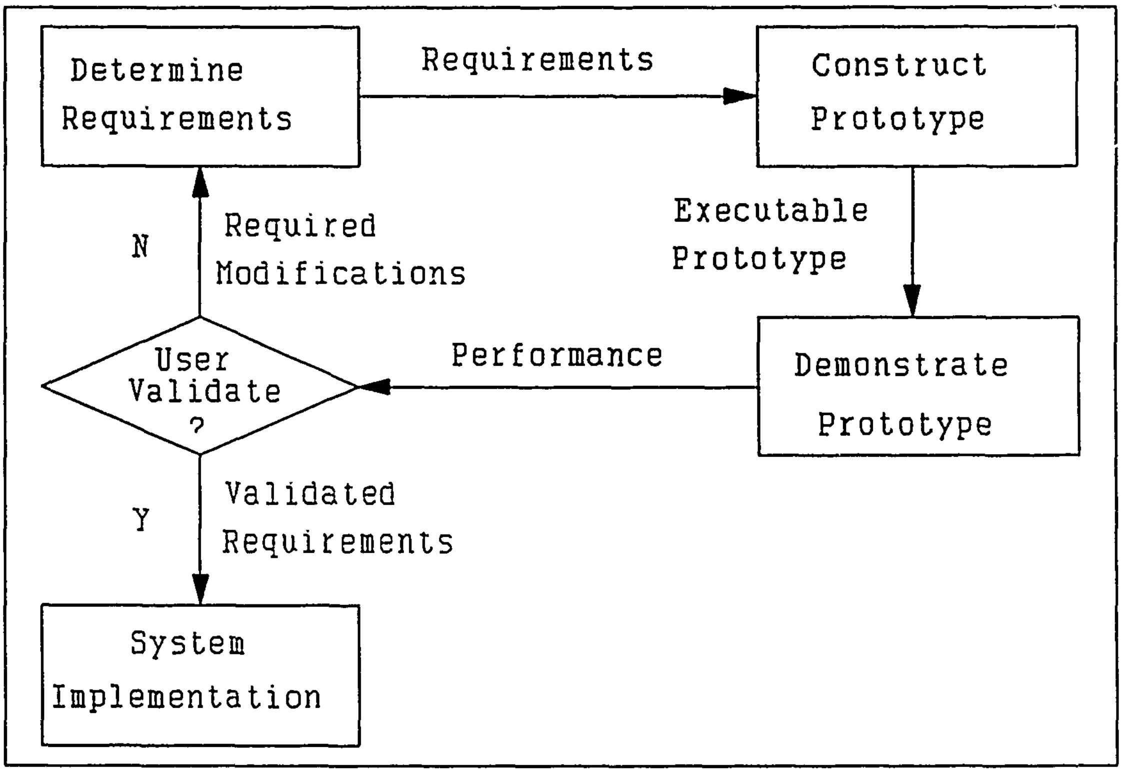


Figure 2 Prototyping Life Cycle

To date, rapid prototyping has been done manually without the aid of software tools. Each step in the rapid prototyping methodology, though faster than the traditional life cycle as discussed above, still requires a good deal of time and effort.[Ref. O'He88 : p. 4]

A computer-aided rapid prototyping approach will provide the software designer with a powerful tool, designed specially for development of hard real-time or embedded systems. Prototyping the system generates a skeletal design framework which may serve as the initial design structure of the production version. The early prototypes provide a traceable link between requirements, design, implementation and maintenance. Figure 3, on page 6, illustrates the major steps in computer-aided prototyping.[Ref. LK88 : pp. 66-72]

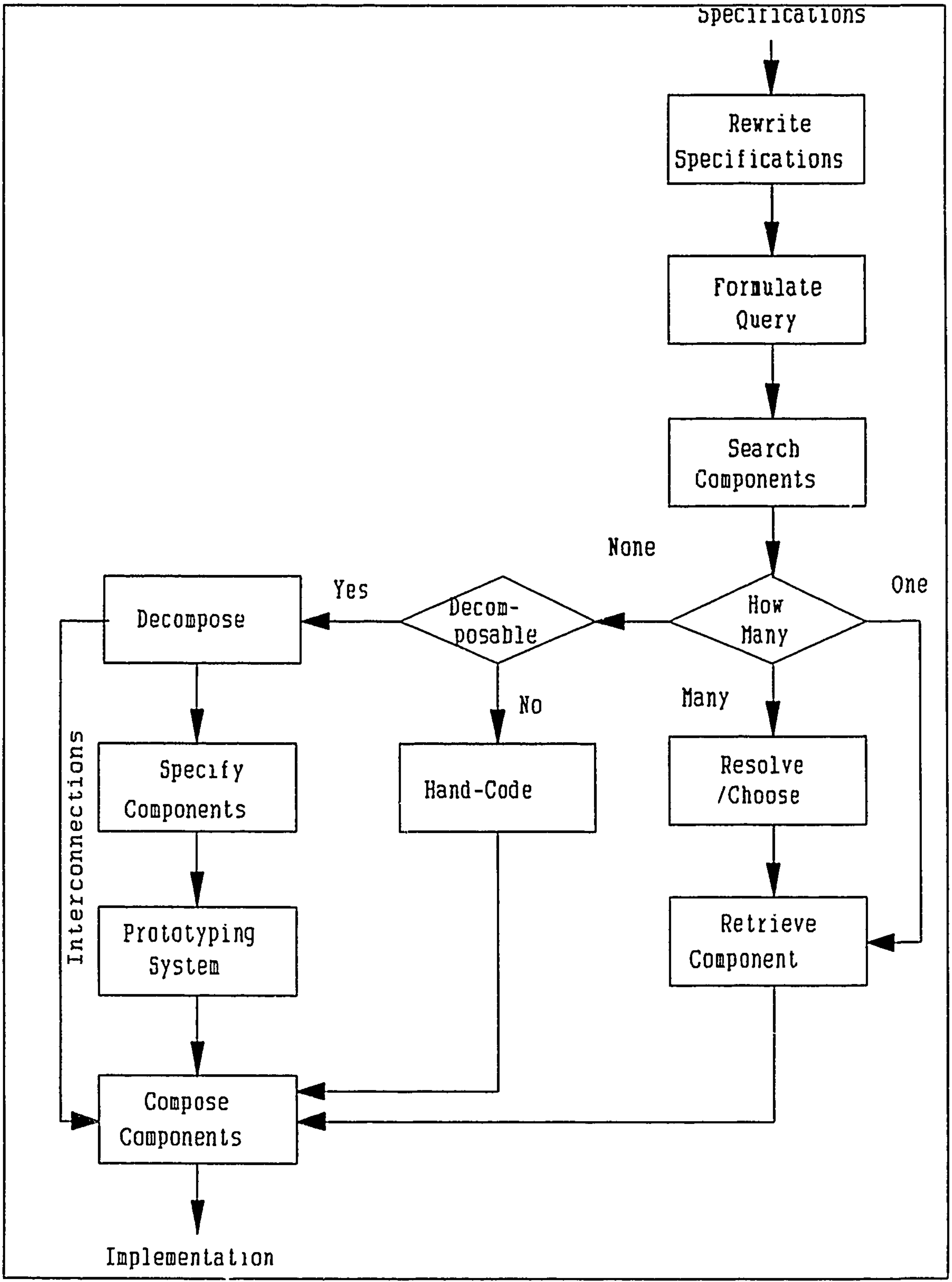


Figure 3 Prototype Development Using the Computer-Aided System

The Computer Aided Prototyping System (CAPS) is being developed to improve software technology, and will aid the software designer in the requirements analysis of large hard real-time systems by using specifications and reusable software components to automate the rapid prototyping process.

The Prototype System Description Language (PSDL) is an executable high level specification language that directly supports CAPS. PSDL is made executable by the execution support system element of CAPS. CAPS and PSDL will be described in detail in the next section.[Ref. LV88 : p. 25-36]

C. CAPS AND PSDL OVERVIEW

1. CAPS

The computer aided prototyping system CAPS consists of three primary subsystems: a user interface, an execution support system, and a prototyping software base. The user interface contributes to effective and efficient construction or modification of prototypes by providing a graphical editor, a syntax directed editor, a browser, an expert system for communicating with end users, and a debugger. The editor enables convenient entry and management of PSDL descriptions and the browser allows the designer to interact with the software database while retrieving and examining prototype components. The expert system provides a paraphrasing capability generating English text from PSDL descriptions. The debugger allows the designer to interact with the execution support system.

The execution support system consists of a translator which generates code to link reusable components together, a static scheduler which allocates time slots for prototype components prior to their execution, and a dynamic scheduler which allocates free time slots to non-time critical components as execution proceeds.

The prototyping database consists of a design database, reusable software base, software design management system and a rewrite system. The prototyping database keeps track of designs and stores reusable prototype components together with their specifications. Its design management system provides version control and maintains design histories, and a retrieval subsystem translates PSDL specifications into a normal form to ease retrieval. Program construction is speeded up by taking advantage of reusable software components drawn from a software base. The aspects of program construction that benefit from automated assistance are retrievals from the software base, generation of code for interconnecting available modules, and static task scheduling. Figure 4, below, graphically describes the major software tools of CAPS, and the Figure 5 on page 9 describes the architecture of CAPS.[Ref. Jan88 : pp. 4-5, and Ref. Luq88 : pp. 14-20]

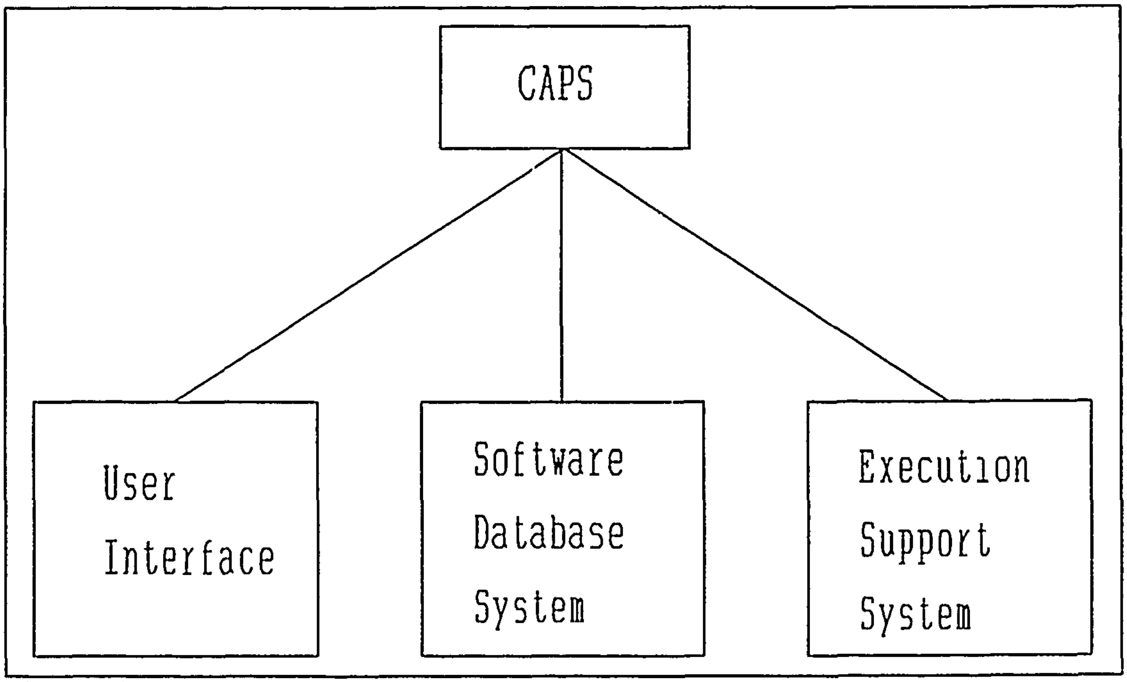


Figure 4 Major Software Tools of CAPS

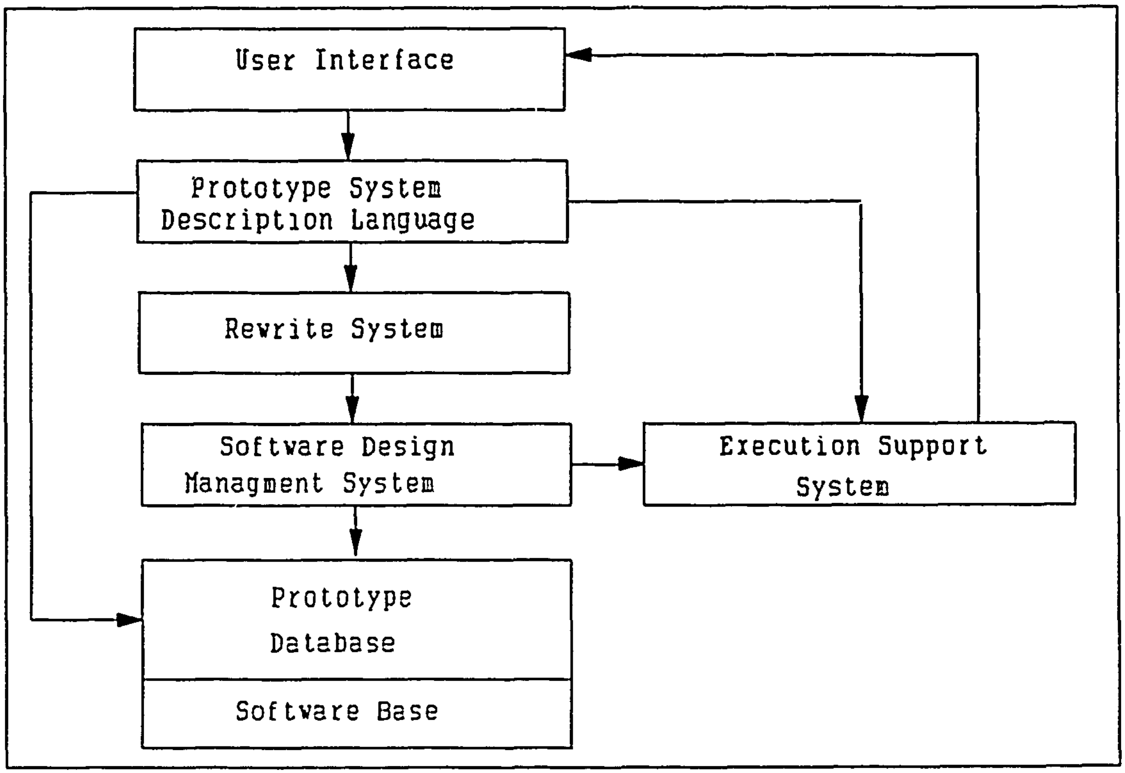


Figure 5 The Computer Aided Prototyping System

2. PSDL

The Prototype System Description Language PSDL was designed to serve as an executable prototyping language working at a specification or a design level. PSDL is a language for describing prototypes of large software systems with hard real-time constraints on different levels of abstraction.

Such systems are modeled in PSDL as networks of operators communicating via data streams, using augmented data flow diagrams. The operators in an augmented data flow diagram are supplemented with timing constraints and non-procedural control constraints. The data stream can carry data values of an abstract data type or tokens representing exception conditions. Each type or operator is either composite or atomic. Composite operators are implemented by decomposing them into networks of more primitive operators using PSDL. Atomic operators are realized by retrieving reusable components from the software base which meets the specifications of operators and are implemented in Ada. The language is easy to use because provides a familiar graphical notation for the underlying computational model. A specification which augments a dataflow graph provides the information to effectively retrieve reusable software components and adapt them to the specific application context.

Computer-aided support of PSDL is provided by an integrated prototyping environment assisting the designer in interactively constructing a PSDL design and automatically links it to reusable components in the software base. The PSDL Execution Support System (ESS) contains a translator, static scheduler, and dynamic scheduler. Figure 6, below, illustrates the ESS subsystems external interfaces to others components of CAPS and the interactions within the ESS itself.[Ref. O'He88 : pp. 7-10]

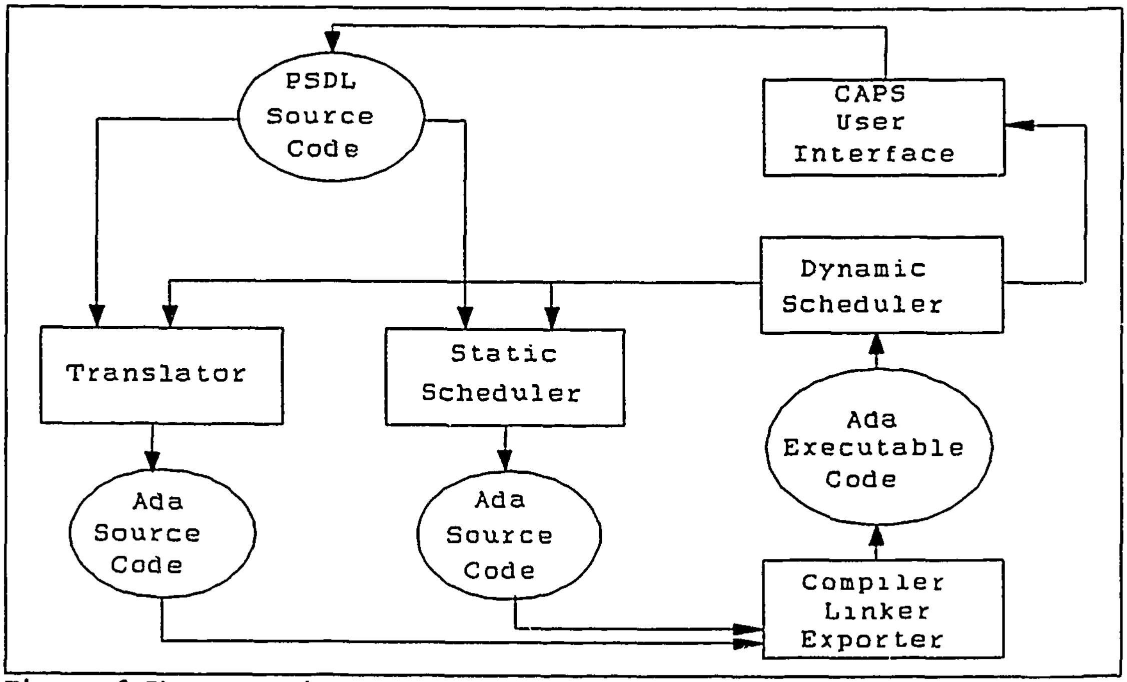


Figure 6 The Execution Support System

D. ORGANIZATION

Chapter II provides a survey of the state-of-the-art in hard real-time scheduling algorithms. Chapter III addresses the design of an optimal scheduling algorithm for handling graph-based hard real-time specifications. The bases for the development of such algorithms are the theories of optimal sequencing through modular decomposition and enumeration techniques. Chapter IV presents the analysis of the optimal scheduling algorithm and establishes its correctness and optimality properties and assess its impacts on the rapid prototyping of hard real-time systems. Chapter V contains the conclusions and recommendations for future work.

II. SURVEY OF PREVIOUS WORK ON HARD REAL-TIME SCHEDULING

The function of a scheduling algorithm is to determine, for a given set of tasks, whether a schedule (the sequence and the time periods) for executing the tasks exists such that the timing, precedence and resource constraints of the tasks are satisfied, and to calculate such a schedule if one exists.

A. SOME DEFINITIONS ABOUT SCHEDULING ALGORITHMS

Task scheduling in hard real-time systems can be static or dynamic. In static systems, a scheduling algorithm determines the schedule for a set of tasks off-line. In dynamic systems, because not all the characteristics of tasks are known a priori, a dynamic scheduling algorithm progressively determines the schedule for tasks on-line. A scheduling algorithm is said to guarantee a newly arriving task if the algorithm can find a schedule for all the previously guaranteed tasks and the new task such that each task finishes by its deadline. A major metric for dynamic scheduling algorithms is the guarantee ratio, which is the total number of tasks guaranteed versus the total number of tasks that arrive.[Ref. BSR88 : pp. 152-160]

A static scheduling algorithm is said to be optimal if, for any set of tasks, it always produces a schedule which satisfies the constraints of the tasks whenever any other algorithm can do so. A dynamic scheduling algorithm is said to be optimal if it always produces a feasible schedule whenever a static scheduling algorithm with complete prior knowledge of all the possible tasks can do so.

Static approaches have low run-cost, but they are inflexible and cannot adapt to a changing environment or to an environment whose behavior is not completely predictable. When new tasks are added to a static system, the schedule for the entire system must be recalculated, which may be expensive in terms of time and money. In contrast, dynamic approaches involve higher run-time costs, but because the way they are designed, they are flexible and can more easily adapt to changes in the environment.

In hard real-time systems, tasks are also distinguished as preemptable and nonpreemptable. A task is preemptable if its execution can be interrupted by other tasks and resumed afterwards. A task is nonpreemptable if it must run to completion once it starts.

B. SOME BASIC TASK DEFINITIONS

A task is a software module that can be invoked to perform a particular function. A task is the scheduling entity in a system. In a hard real-time system, a task is characterized by its timing constraints, precedence constraints, and resource requirements. This thesis assumes that the resource requirements are always met.

The precedence constraints among a set of tasks specify the relations between the tasks. A task T_i is said to precede task T_j if T_i must finish before T_j begins. Interrelated tasks communicate with each other in real-time to achieve synchronization as well to exchange data. The precedence graph of a set of tasks is an acyclic graph.

C. DESCRIPTION OF SOME SCHEDULING ALGORITHMS

In this section we survey both static and dynamic scheduling algorithms for hard real-time systems. However, because of the enormous amount of literature which deals with hard real-time scheduling problems, it is imp.ssible to discuss all the material. Therefore, we only present an overview of previous work scheduling algorithms approaches and discuss their characteristics.

1. The Fixed Priorities Scheduling Algorithm

In many conventional hard real-time systems, tasks are assigned with fixed priorities to reflect critical deadlines, and tasks are executed in an order determined by the priorities. During the testing period, the priorities are (usually manually) adjusted until the system implementer is convinced that the system works. Such approach can only work for relatively simple systems, because it is hard to determine a good priority assignment for a system with a large number of tasks by such a test-and-adjust method. Fixed priorities is a type of static scheduling. Once the priorities are fixed on a system is very hard and expensive to modify the priority assignment.[Ref. LTJ85]

2. The Harmonic Block with Precedence Constraints Scheduling Algorithm

This scheduling algorithm is being used by the CAPS. A general description of the implementation is furnished above, and a data flow diagram is given in Figure 7, on page 15.

The first component of the DFD, "Read_PSDL", reads and processes the PSDL prototype program. The output of this step is a file containing operator identifiers, timing information and link statements.

The second component is the "Pre-Process_File". The file generated in the first step is analyzed and the data is divided into three separate files based on its destination or additional processing required. The Non-Crits contains the data of all noncritical operators for use by the dynamic scheduler. The Operator file contains all critical operators identifiers and their associated timing constraints. The Links file contains the link statements which syntactically describe the PSDL implementation graphs. During this step some basic validity checks on the timing constraints are performed, if any of the checks fails an exception is raised and an appropriate error message is submitted to the user.

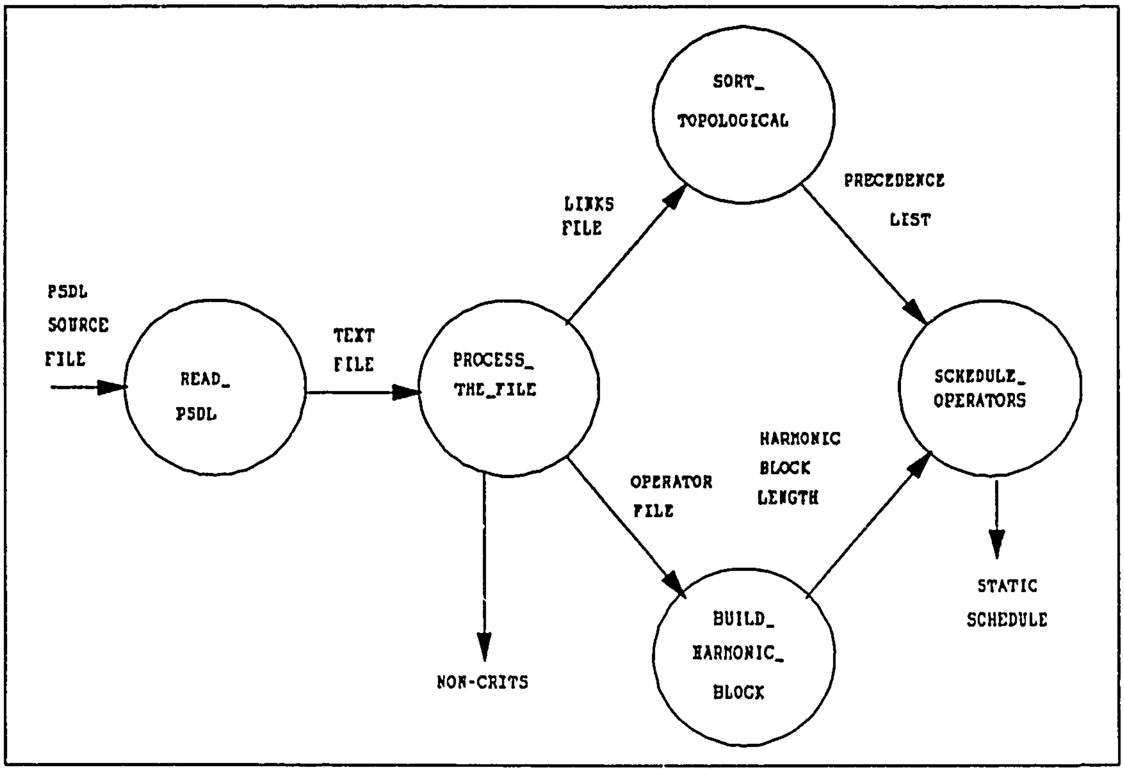


Figure 7 1st Level DFD

The "Sort_Topological" component performs a topological sort of the link statements contained in the Links file. The requirements for a topological sort imply that the statements being sorted have natural continuity and connectedness. These properties define the execution precedence of the time critical operators regardless of whether the graphs are linear or acyclic. The output from the topological sort is a precedence list of critical operators stipulating the exact order in which they must be executed. A linear graph will produce one precedence list while an acyclic graph can produce two or more different precedence lists.

The second output of the "Pre-Process_File", the Operator file, is the input to the "Build_Harmonic_Blocks". An harmonic block is defined as a set of periodic operators where the periods of all its component operators are exact multiples of a calculated base period. Each harmonic block is treated as an independent scheduling problem. When multiprocessors are utilized then one processor for each harmonic block is necessary. The implementation being developed utilizes a single processor, therefore the final static schedule assumes that only one harmonic block is created. All the operators must be periodic, then all the sporadic operators are converted to their periodic equivalents. The periodicity helps insure that execution is completed between the beginning of a period and its deadline, which defaults to the end of the period.

In order to convert a sporadic operator into its equivalent periodic operator the following parameters of the sporadic operator must be known :

- Maximum Execution Time (MET).
- Minimum Calling Period (MCP).
- Maximum Response Time (MRT).

Some rules must be obeyed by these parameters in order to cbtain an equivalent periodic operator, the rules are the following:

- MET < MRT. This rules insures that (MRT - MET) produces a positive value.
- MCP < MRT. This condition is necessary, but not sufficient, to guarantee that an operator can fire at least once before a response is expected.
- MET < MCP. This restriction insures that the period calculated will conform to a single processor environment.

The periodic equivalent is then calculated as P = min (MCP, MRT - MET). The value of P must be greater than MET in order for the operator to complete execution within the calculated period.

After all the operators are in periodic form, they are sorted in ascending order based on the period values. A second preliminary step is to calculate the base block and its period for the sorted sequence of operators. The base period is defined as the greatest common divisor (GCD) of all the operators in one sequence that will be scheduled together.

The last preliminary step is to evaluate the length of time for the harmonic block. The actual harmonic block length is the least common multiple (LCM) of all the operators' period contained in the block. The harmonic block and its length are an integral part of the static schedule. This block represents an empty timeframe within which the operators will be allocated time slots for execution. The allocation of time slots within the harmonic block is repeated indefinitely.

The outputs of the Sort_topological" and the "Build_Harmonic_Blocks" are used by the "Schedule_Operators" in order to create a static schedule for the time critical operators. The resulting static schedule is a linear table giving the exact execution start time for each critical operator and the reserved MET within which each operator completes its execution.

This linear table is evaluated in two iterative steps. In the first step an initial execution time interval is allocated for each operator based on the equation INTERVAL = (current time, current time + MET). Next the process creates a firing interval for each operator during which the second iterative step must schedule the operator. The firing interval stipulates the lower and upper bound for the next possible start time for an operator based on its period. The second step, initially, uses the lower bound of each firing interval when it schedules operators during subsequent iterations. The sequence of operators is allocated time slots according to the earliest lower bound first. Before an operator is allocated a time slot, this step verifies that:

- (current time + MET) =< harmonic block length.

This condition is applicable to every operator scheduled in that harmonic block. This step also calculates new firing intervals for each operator scheduled. Once all the operators are correctly scheduled within an entire harmonic block a static schedule is available. All subsequent harmonic blocks are copies of the first.

A theoretical development and implementation guideline of this algorithm is available in [Ref. O'He88] and [Ref. Jan88].

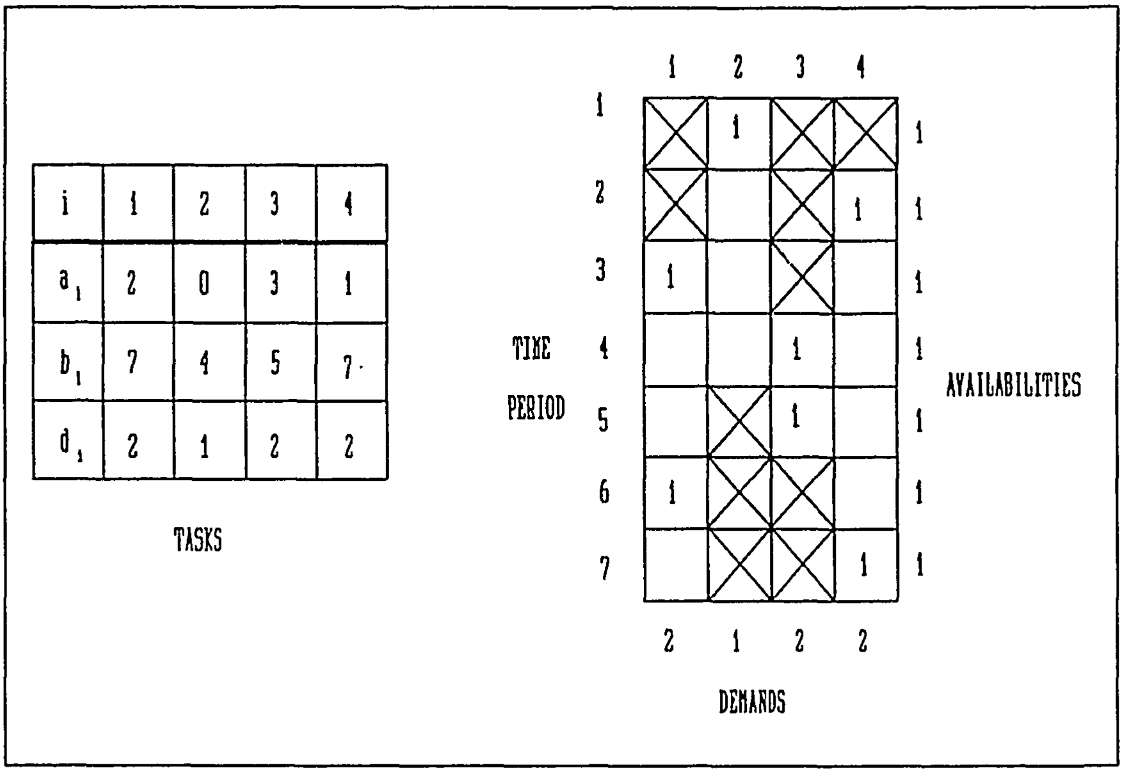
The actual implementation of this algorithm and the analysis of its performance is described in [Ref. Mar88].

3. The Earliest Start Scheduling Algorithm

This algorithm considers the scheduling of n tasks on a single processor, additional constraints that each task has an earliest start time (a_i). Each task becomes available for processing at time a_i, must be completed by time b_i, and requires d_i time units for processing. Task splitting (preemptable tasks) is allowed. Under this assumption it is only required to complete d^k_i units of processing between a_i and b_i.

Consider the rectangular matrix that has a column for each job and a line for each unit of time available. There are $\max_i(b_i)$ lines and n columns. In this matrix it is necessary to distinguish between admissible and inadmissible cells. For job i the cell (i,j) is admissible if $a_i<j=<b_i$ and inadmissible otherwise. The admissible cells correspond to the time where the task may be performed. Figure 8, on page 19, shows an example.

Associated with each row is an availability of one unit of time, and with each column a requirement of d_i. If the task i is being processed at time j, a 1 is placed in the admissible cell. This problem is equivalent to that of finding a set of 1's placed in admissible cells such that columns sums satisfy the requirements d_i and each line contains at most one single 1.[Ref. BFR71 : pp. 511-514]



i	1	2	3	4
a_i	2	0	3	1
b_i	7	4	5	7
d_i	2	1	2	2

TASKS

TIME PERIOD	1	2	3	4	AVAILABILITIES
1	☒	1	☒	☒	1
2	☒		☒	1	1
3	1		☒		1
4			1		1
5		☒	1		1
6	1	☒	☒		1
7		☒	☒	1	1
DEMANDS	2	1	2	2	

Figure 8 Earliest Start Time Scheduling

This type of algorithm does not account for precedence constraints. In order to include the precedence constraints in this algorithm it is necessary to do some modifications. The modification can utilize concepts like the harmonic block and the restriction that a job j, that is preceded by a job i, is admissible only after the constraint a_i is satisfied.

The [Ref. BFR71 : pp. 518-519] presents an implementation in FORTRAN to solve the case without precedence constraints. This type of algorithm ıs not applicable to our case because it assumes that all the tasks are preemptable.

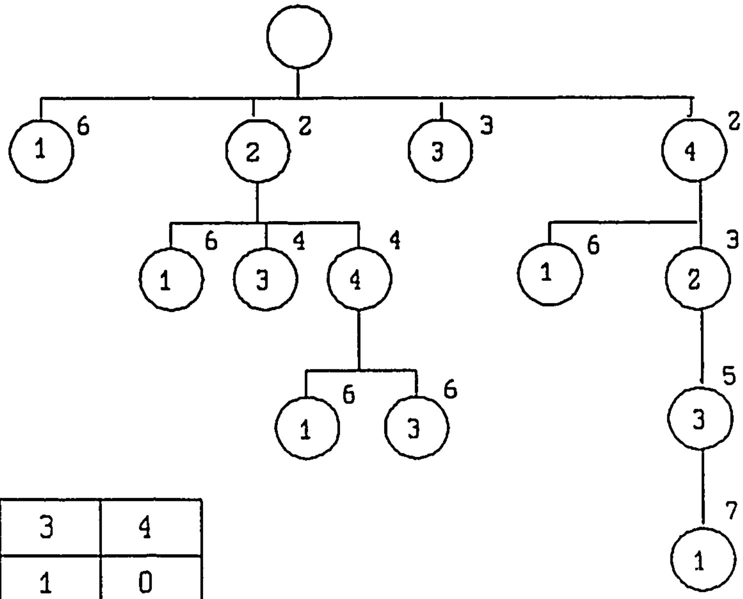
This algorithm assigns a time slot to the newest ready task to be executed, if this allocation will imply in to miss the deadline of some task already started then the algorithm will assign the time slot to the task with the nearest deadline. If there is no new task ready then the next time slot is assigned to the task with the nearest deadlıne.

This algorithm is bounded by O(n) in time, and does not guarantee that a solution (assuming that at least one is available for the problem) is found.

4. The Branch and Bound Scheduling Algorithm

This section covers the scheduling of n tasks on a single processor under the assumption that job splitting is not allowed. The notation used in this section is the same as the section II.C.3. The main ideia is to enumerate implicitly all the possible orderings by a branch, exclude and bound algorithm. In this approach the precedence constraints are not included in the analysis, but they may be easily taken into account during the branch step. During the branch all infeasible sequences due to violation of the due date are discarded (here is possible to include the precedence constraints).

All the possible sequences are enumerated by a tree type construction, as shown in Figure 9, on page 21. From the initial node we branch to n new nodes on the first level of descendent nodes. Each of these nodes represents the assignment of task i, 1 <= i <= n, to be the first in the sequence (the number inside the node represents the task). Associated with such a node there is the completion time t^{ij}, of the task j in the position i, i.e., $t^{1i} = a_i + d_i$ (the completion time of each task, in a given branch of the tree, is indicated by the number outside the correspondent node, the number inside the node represents the task being assigned). Next we branch from each node on the first level to (n-1) nodes on the second level. Each of these nodes represents the assignment of each of the (n-1) unassigned tasks to be second on the sequence. As before, we associate the corresponding node the completion time of the task $t^{2j} = \max(t^{1i}, a_j) + d_j$, where i is the parent of the task j in the branch being evaluated. We continue in similar fashion. The initial node is a dummy node, in the unconstrained case all the node must be present in the level 1 (level 0 is assumed to be the dummy root of the complete tree), in case with precedence constraints in the level 1 we allocate only the tasks that have only external input or no predecessor.



Tasks

i	1	2	3	4
a_i	4	1	1	0
b_i	7	5	6	4
d_i	2	1	2	2

Figure 9 Branch and Bound Scheduling

Consider the (n-k+1) new nodes generated at the level k of the tree construction, if the finish time t^{ki} associated with at the least one of these nodes exceeds its due date then the subtree rooted at each one of the nodes that are unfeasible may be excluded from further consideration.

The bounding condition applies only when there are no precedence constraints and is intended to find an optimal (minimizing the length of the block) ordering of the sequence.

In the case with precedence constraints this algorithm guarantees an optimal solution, one disadvantage is the time complexity which is factorial in the number of tasks in the worst case. A more detailed explanation, as well a step by step definition of the algorithm, may be found in [Ref. BFR71 : pp. 514-519].

Another possible implementation of this algorithm is to utilize the concepts: length of the harmonic building block, and the firing interval for each task; described in the previous algorithm. In order to include the precedence constraints and the period of the operators the following scheme variant was developed:

- define the agenda list as an empty list, define the waiting set as an empty set,
- define the successors and predecessors (the precise definition of these terms is described in section III.D.1) of each task,
- evaluate the length of the harmonic block,
- select the tasks that have no predecessors and put them in the waiting set,
- select the task from the waiting set that has the smallest earliest start time (ties may occur, then some other criteria must be applied), if all the predecessors of this task are in the agenda list then put this task in the agenda list as the last component, put all the successors of this task in the waiting list; otherwise select the next task with smallest earliest start time,
- evaluate the next firing interval of the task selected, if it is greater than the length of the harmonic block then remove this task from the waiting list; if the task was removed from the waiting list then verify if the waiting list is empty, if it is empty then stop the agenda list contains the schedule, otherwise go to the previous step.

The algorithm described above is not optimal as is the branch and bound tree described in the reference, but has the advantage that is more compact in time and space. The main deviation of the algorithm described above from the idea expressed in the paper is that this algorithm does not take in account all the possible branches, when a decision about more than one branch must be done then after this point is not possible to come back and test the other branches. Another possible version of this algorithm is to consider as criteria for inclusion in the agenda list the earliest deadline instead the earliest start time, the description of the algorithm needs to be slightly modified. An implementation of the variant described above is available in [Ref. Kil89].

5. The Minimize Maximum Tardiness with Earliest Start Scheduling Algorithm

This algorithm considers a sequencing problem consisting of n tasks and a single processor. Task i is described by the following parameters:

- the ready time (a_i), the earliest point in time at which processing may begin on i (i.e., an earliest start time).
- the processing time (d_i), the interval over which task i will occupy the processor.
- the due date (b_i), the completion deadline for task i.

The three characteristics a_i, b_i, and d_i are known in advance and no preemption is allowed in the processing of the tasks.

As a result of scheduling, task i will be completed at time C_i and will be tardy if $C_i > d_i$. The tardiness of task(T_i) is defined by $T_i = \max\{0, C_i - d_i\}$. The scheduling objective is to minimize the maximum task tardiness, which is simply $T_{max} = \max_j \{ T_j \}$.

For the static version of the n tasks single processor problem without precedence constraints(all a_i's equal), T_{max} is minimized by the sequence $b_{(1)} =< b_{(2)} =< \ldots =< b_{(n)}$, that is, by processing the tasks in nondecreasing order of their deadlines.[Ref. BS74 : pp. 172]

In the dynamic version of the problem the statement above can also be applied if the tasks can be processed in a preemptable fashion, in this case sequencing decisions must be considered both at task completion and at task ready time. Then we have the following:

- At each task completion the task with minimum b_i among available tasks is selected to begin processing.
- At each ready time, a_i, the deadline of the newly available task is compared to the deadline of the task being processed. If b_i is lower, task i preempts the task being processed otherwise the task i is simply added to the list of available tasks.

The solution to the preemptive case is not difficult to construct because the mechanism is a dispatching procedure. Since all nonpreemptive schedules are contained in the set of all preemptive schedules, the optimal value of T_{max} in the preemptive case is at least a lower bound on the optimal T_{max} for the nonpreemptive schedules. This principle is the basis for the algorithm.

In the nonpreemptive problem, there is a sequence corresponding to each permutation of the integers 1, 2, ..., n. Thus there are at must n! sequences, but some of these sequences do not need to be considered. The number of feasible sequences depends on the data in a given problem, but will be usually less than n!.

A branch and bound algorithm will be used to systematically enumerate all the feasible permutations.

The branching tree is essentially a tree of partial sequences. Each node in the tree at level k corresponds to a partial permutation containing k tasks. Associated with each node is a lower bound on the value of the maximum tardiness which could be achieved in any completion of the corresponding partial sequence (obtained using the preemptive adaptation). The calculation of lower bound allows the algorithm to enumerate many sequences only implicitly. If a complete sequence has been found with a value T_{max} less than or equal to the bound associated with some partial sequence, then it is not necessary to complete the partial sequence in the search for optimum solution.

The branch and bound algorithm maintains a list of nodes ranked in nondecreasing order of their lower bounds. At each stage the node at the top of the list is removed and replaced on the list by several nodes corresponding to augmented partial sequences. These are formed by appending one unscheduled task to the removed partial sequence. The algorithm terminates when the node at the top of the list corresponds to a complete sequence. At this point, the complete sequence attains a value of T_{max} which is less than or equal to the lower bound associated with every partial sequence remaining on the list, and the complete sequence is therefore optimal.

Before the tree search begins, the algorithm uses a heuristic initial phase to obtain a feasible solution to the problem. This initial feasible solution allows the tree search to begin with a complete schedule already on hand, and allows several partial schedules to be discarded in the course of the tree search, simply because their bound exceed the value of the initial solution.

There are four heuristic available:

- Ready time: sequence the tasks in nondecreasing order of their ready time, a_i.
- Deadline: sequence the tasks in nondecreasing order of their deadlines, b_i.
- Midpoint: sequence the tasks in nondecreasing order of the midpoints of their ready times and deadlines $(a_i + b_i)/2$. Hence use the nondecreasing order of $a_i + b_i$.
- PIO: sequence the tasks in the order of their first appearance in the optimal preemptive schedule, which is constructed by the dynamic version.

The [Ref. BS74 : pp. 171-176] contains a complete and detailed description of the algorithm as also an analysis of the performance of the algorithm considering each heuristic, the global time complexity of this algorithm is $O(n^2)$.

In [Ref. Hor74 : pp. 177-185] we may find some simple and quick algorithms for the same set of conditions.

As can be visualized this algorithm does not take into account the possible precedence constraints among the tasks, these precedence constraints must be take in account during the evaluation of the branch and bound solution of the tree search. The inclusion of the precedence constraints in the evaluation of the heuristics must also be considered. The algorithm can be extended to handle the case where tasks can be started only after some instance of time in the future (this happens when some of the tasks are periodic), the modification necessary is in the definition of task's scheduled start time.

6. The Deadline and Criticalness Scheduling Algorithm

This algorithm is based upon the following assumptions:

- All application tasks are known,but their invocation order is not known. That is, tasks arrive dynamically and independently.
- There are no precedence constraints on the tasks; they can run in any order relative to each other as long deadlines are met.
- Each task has the following characteristics: an arrival time (a_i) that is the time at which the task is invoked; a worst-case computation time (d_i) that is the maximum time needed for it completion; a criticalness (n_i) that is one of the n possible levels of importance of the task; a deadline (b_i) that is the time by which the task has to complete execution. These characteristics are time invariant.

The algorithm that will be discussed in this subsection assumes the existence of an environment that consists of a distributed system consisting of N nodes. Each node contains m processors divided into two types: systems processors dedicated to executing system tasks and application processors executing only application tasks. The connection medium for the nodes is assumed to be a shared bus. In other words the system under analysis consists of a collection of multi-processors connected together in a loosely-coupled network.

The main systems of interest to the discussion are the local scheduler and the global scheduler. The local scheduler at each node maintains a data structure called the System Task Table (STT); this table contains a list of applicable tasks that have been dynamically guaranteed to make their deadline at this local node. Entries in the STT are arranged in the order of execution and tasks are dispatched for execution from this table. Each STT entry, corresponding to a guaranteed task, has five attributes: the arrival time, the latest start time, the criticalness, the deadline, and the computation time.

The Local Scheduler, which can re-order, insert or remove any entries in the STT, is activated upon the arrival of a new task at the local node, or in response to a bidding process which is initiated by the global scheduler. The Local Scheduler, working in a copy of the STT, determines if a new task can be inserted into the current STT such that all previous tasks in the STT as well as the new task meet their deadlines. If so then the task is guaranteed and the latest start time is determined. If the new task cannot be guaranteed locally, or can only be accommodated at the expense of some previously guaranteed task(s), then the rejected task(s) is (are) handed over to the Global Scheduler.

The Global Scheduler then takes the necessary actions to transfer the task(s) to any alternative nodes that may have the resources to accept this (those) task(s). The Global Scheduler uses bidding. Request-for-bids (RFB) are broadcast to the other nodes when a local task has to be reallocated. If several remote nodes respond with bids reflecting their surplus, the Global Scheduler evaluates those bids and transfers the task to the node with the best bid.

The algorithm first attempts to guarantee an incoming task according to its deadline, ignoring its criticalness. If the task is guaranteed then the scheduling is successful. However, if this first attempt at scheduling fails, then there is an attempt to guarantee the new task at the expense of previously guaranteed, but less critical tasks. If enough less critical tasks can be found then the new task is guaranteed at this site and the removed tasks are transferred to alternative sites. If there are not enough less critical tasks, or the deadline of the new task is such that the removal of any such tasks does not allow the new task to meet its deadline, then the new task is transferred to an alternative site. The process is repeated at the next node until the task either meets its deadline or its deadline expires.

A detailed explanation of the algorithm above, discussing all the steps as well the performance is contained in [Ref. BSR88 : pp. 152-160].

7. The Rate-Monotonic Priority Assiggnment Scheduling Algorithm

This algorithm assumes the following premises:

- The requests for all the tasks for which hard deadlines exist are periodic, with period (p_i).
- Deadlines consist of run-ability constraints, that is each task must be completed before the next request for it occurs.
- The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks.
- Run-time for each task is constant (d_i) and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption.

An important concept in determining the rule is that of the *critical instant* for a task. The *deadline* of a request for a task is defined to be the time of the next request for the same task. The *response time* of a request for a certain task is defined to be the time span between the request and the end of the response to that request. A *critical instant* of a task is defined to be an instant at which a request for that task will have the largest response time. A *critical time zone* for a task is the time interval between a critical instant and the end of the response to the corresponding request to the task.

Based on the definitions above is possible to infer that a critical instant for any task occurs whenever the task is requested simultaneously with requests for all higher priority tasks. One of the values of this result is that a simple direct calculation can determine whether or not a given priority assignment will yield a feasible scheduling algorithm. Specifically, if the requests for all tasks at their critical instants are fulfilled before their respective deadlines, then the scheduling algorithm is feasible. As an example consider two tasks T_1 and T_2 with $p_1 = 2$, $p_2 = 5$, and $d_1 = 1$, $d_2 = 1$. If we let T_1 be the higher priority task then from Figure 10 (a), on page 28, we see that such priority assignment is feasible. Moreover, the run time of T_2 can be increased at most to 2 but not further as illustrated in Figure 10 (b). On th other hand, if we let T_2 be the higher priority task, then neither of the values of d_1 and d_2 can be increased beyond 1 as illustrated in Figure 10 (c).

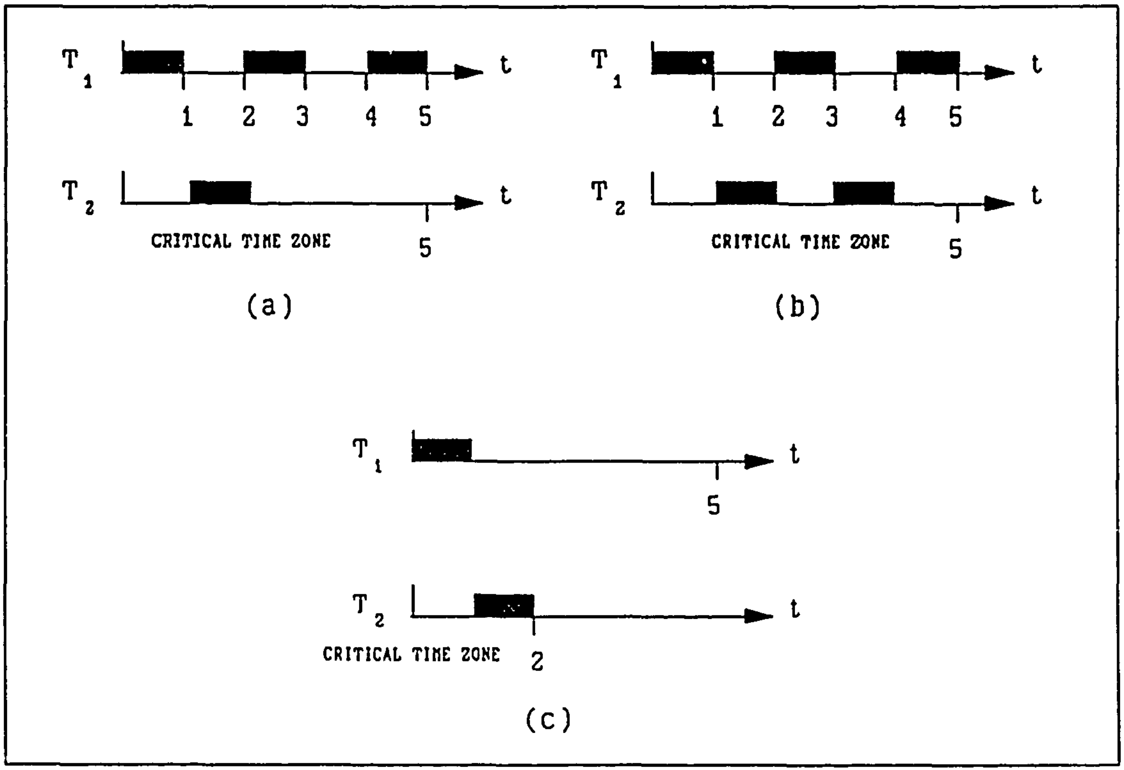


Figure 10 Schedule of Two Tasks

The analysis of the example above suggests a priority assignment. Let p_1 and p_2 be the request periods of two tasks, with $p_1 < p_2$. If we let T_1 be the higher priority task then, according to the definition of critical instant, the following inequality must be hold $\lfloor p_2/p_1 \rfloor d_1 + d_2 =< p_2$[1].

If we let T_2 be the higher priority task, then, the following inequality must be satisfied $d_1 + d_2 =< p_1$. In other words, whenever the $p_1 < p_2$ anr d_1, d_2 are such that the task schedule is feasible with T_2 at higher priority than T_1, it

[1] This condition is necessary but not sufficient to guarantee the feasibility of the priority assignment. The symbol $\lfloor x \rfloor$ denotes the largest integer smaller than or equal to x.

is also feasible with T_1 at higher priority than T_2, but the opposite is not true. Thus we should assign a higher priority to T_1 and lower priority to T_2. Hence, more generally, it seems that a reasonable rule of priority assignment is to assign priorities to tasks according to request rates, independent of their run-times. Specifically, tasks with higher request rates will have higher priorities. Such an assignment of priorities is known as the Rate-Monotonic Priority Assignment. Such priority assignment is optimum in the sense that no other fixed priority assignment rule can schedule a task set which cannot be scheduled by the rate-monotonic priority assignment.

A formal development and analysis of this algorithm, as well the theoretical development of maximum achievable processor utilization of this type of algorithm is available in [Ref. LL72 : pp. 46-61].

Some algorithms for scheduling periodic tasks to minimize average error utilize the rate-monotonic priority assignment algorithm in order to solve the scheduling of the mandatory part of all the tasks, a complete description of these algorithms may be found in [Ref. CL88 : pp. 142-150].

8. The Priority Ceiling Protocol Scheduling Algorithm

The priority ceiling protocol[2] is based upon the rate-monotonic priority assignment. It minimizes the problem of problem of priority inversion in the presence of resource constraints. Priority inversion is any situation in which a lower priority task holds a resource while a higher priority task is ready to use it.

This protocol assumes the use of binary semaphores to synchronize access to shared data. The main idea is to represent each semaphore as a server task. Each critical region is represented as an entry of the task. Server tasks are

[2] This protocol is intended to be used when developing Hard real-time system using Ada.

the only form of task allowed to contain an accept statement. A client task is a non-server task that contains at least one entry call. A server task is said to executing on behalf of client task T if the server has been called either by T or by a server task that is executing on behalf of T . The priority ceiling of a server task is defined as the highest priority of its clients tasks, i.e., the highest priority of task that has called the server directly or indirectly.

The main idea behind this scheme is to dynamically increase the priority of the server task to the value of the priority ceiling to avoid priority inversion.

To apply the priority ceiling protocol in Ada, the following restrictions on the use of Ada tasking features must be obeyed:

- All accept statements in a task must be contained in a single select statement that is the only statement in the body of an endless loop. There must be no guards on the select alternatives and no nested accept statements. A task that contains such an accept is called a server task.
- There must be no conditional or timed entry calls.
- Each task must assigned a priority.
- A server task must have a priority lower than any of its clients tasks.

Under these conditions and definitions, the ceiling protocol priority guarantees that a set of n periodic non-server tasks can be scheduled by the rate-monotonic algorithm if the following conditions are satisfied:1 =< i =< n, and $(d_1/p_1) + (d_2/p_2) + \ldots + (d_i/p_i) + B_i/p_i =< i\ (2^{1/i} - 1)$, where d_i is the execution time of non-server task T_i, p_i is the period of the non-server task T_i, and B_i is the worst case blocking time of the non-server task T_i.

Another algorithm very similar to this one is the priority inheritance protocol, but the priority inheritance protocol has a performance that is lower than the performance of the priority ceiling protocol.

An extensive comparison between both algorithms is done in [Ref. LSS88], a complete discussion of the priority ceiling protocol is available in [Ref. Sha88].

9. The Bandwidth Preserving Scheduling Algorithms

These types of algorithms try to solve the deficiency of the rate-monotonic approach of not applying to sporadic tasks. When the sporadic tasks are critical, they can be incorporated into the rate-monotonic approach through the use of a periodic server, a periodic task whose function is to service one or more sporadic tasks. This is the pooling approach commonly used to provide predictable sporadic response times.

These algorithms are classified as bandwidth preserving since they can overcome the limitation of polling where the sporadic task arrives after the polling instant. The algorithms are termed Priority Exchange and Deferrable Server and are explained below.

a. *Priority Exchange Algorithm*

This algorithm may best be described by using an example. Consider a set of n periodic tasks, T_1 to T_n with run-times, d_1 to d_n, assigned priority by the rate-monotonic algorithm. Let periodic server T_1 be used to service sporadic requests. The Priority Exchange algorithm allows the highest priority periodic server T_1, to exchange high priority run-time d_1, for lower priority periodic task run-time T_i for i greater than or equal to 2. The algorithm works as follows: T_1 will always use its high priority run-time if there are sporadic requests pending. If there are no sporadic requests pending and there are periodic requests pending, T_1 will trade its high priority run-time d_1 for the highest priority pending periodic task T_i's run-time d_i until it has exhausted all its high priority run-time or sporadic requests arrive at which point it uses its remaining run time to service the sporadic requests. The run-time of the periodic task is advanced thus maintaining its schedubility and the periodic server's run-time is now at low priority. Since the object is to maximize sporadic response time without endangering periodic deadlines, and distinct tasks may have the same priority level (always smaller than the sporadic task server), ties are broken in favor of the sporadic task server.

The only case where the periodic server T_1 must completely sacrifice his run-time is when the resource is idle, that is when there are no sporadic or periodic tasks pending.

b. Deferrable Server Algorithm

The Deferrable Server algorithm is similar to the Priority Exchange algorithm but easier to implement. Unlike the Priority Exchange algorithm the Deferrable Server algorithm does not trade down its high priority run-time d_1 when there are no sporadic tasks pending but rather holds its high priority run-time until the end of the server period. The cost of this reduced complexity is a slight decrease in the worst case periodic task scheduling.

The Deferrable Server algorithm creates a periodic server T_1 with d_1 run-time with priority defined by the server's period T_1. This server has the entire period within which to use its d_1 run-time at priority Pr_1. If at the end of the period any portion of the d_1 run-time is not used then it is discarded.

The formal proof of the feasibility of the schedule generated as well a detailed analysis of these algorithm can be found in [Ref. LSS88]. An improved version of the Priority Exchange algorithm is available for analysis in [Ref. LSS88 : pp. 251-258].

10. The Time-Driven Systems using Augmented Petri Nets Model

This model outlines a methodology for specifying the timing requirements for a class of time-driven embedded systems. In this model time-driven systems are defined as systems wherein the time in which the system and portions of the system execute their intended functions is critical to successful performance, and wherein a master timing mechanism controls the repetitive performance of similar activities at regular intervals. This means that we are working with hard real-time systems with periodic tasks and precedence constraints.

The approach taken is to use a Petri net[3] to model a time-driven system. The Petri net model is then augmented by attaching an execution time variable to each node in the network representing a task in the system. Although most of the work in this area indicates that the notion of time may be included as a part of a procedure attached to the nodes used to model the system, the concept is not fully developed.[Ref. CR83 : pp. 603-616]

A Petri net is a bipartite directed graph consisting of place nodes and transition nodes. Places, drawn as circles, are used to represent conditions; transitions, drawn as bars, are used to represent events. The "marking" (m) of a Petri net is a function that assigns tokens to the places of the net. Tokens, drawn as small dots in the circles, are used to define the execution state of the Petri net, and their number and position change during execution.

The marking of a Petri net is changed by the firing of transitions. A transition is enabled to fire if and only if there is at least one token in each of its input places. The firing of a transition is an instantaneous event during which one token is removed from each of the transition's input places and one token is deposited in each of its output places.

The removal of tokens from input places as result of transition firing has the effect that, if two or more transitions are currently enabled by the presence of a token at the same input place, the firing of any one of those transitions removes that token and disables the remaining transitions. These transitions are

[3] A good introduction, as well an extensive bibliography about Petri Nets, is available in the paper *"Petri Nets"*, J.L. Peterson, ACM Computing Surveys, Vol. 9, No. 3, September 1977, p. 224-252.

said to be in conflict, and the place causing the conflict requires a decision to be made between multiple output paths. Figure 11, below,shows a Petri net with three transitions firing consecutively.

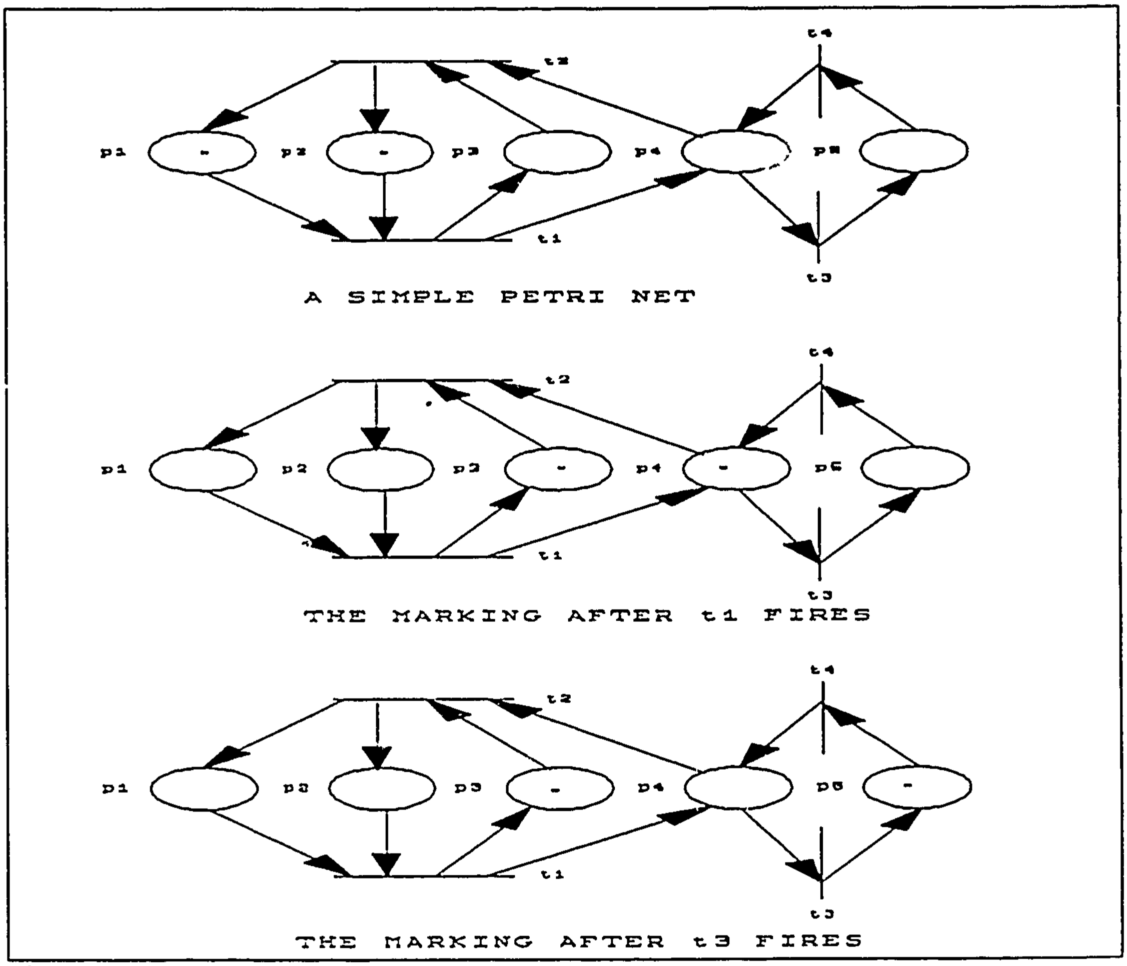


Figure 11 A Petri Net

The places are used to represent tasks, and the execution of a task is modeled by a transition representing the instantaneous start of execution with a directed arc to a place representing the condition of that task being in execution. A nonnegative execution time d_i is assigned to each place p_i. A token becomes ready to aid in enabling an output transition of place p_i only after d_i

time units have expired since p_i first received the token. This modeling approach preserves the classic Petri net notion of transitions as a instantaneous events, and it does not obscure the state of the system during the time that a task is in execution.

The net constructions to be used are best defined in terms of input and output functions of their transitions and places. Let I_t be a set-valued transition input function mapping a transition t_i to the set of places from which arcs exist to t_i. Similarly, let O_t be a set-valued transition output function mapping t_i to the set of places to which arcs exist from t_i. The input and output functions may be extended to include a similar place input function (I_p) and a place output function (O_p).

To model time in a Petri net a basic construction will be defined:

- The master timing mechanism is modeled by a net construction that includes a cycle, called the driving cycle because its execution time drives the execution time of the remainder of the Petri net. The master timing mechanism consists of a place p_1, the master timing task, connected by an elementary loop to a transition t_1 such that : the initial marking of p_1 ($m_1=1$) reproduces itself with a fixed execution time (driving cycle time) T_1; I_t $(t_1) = \{p_1\}$, t_1 only input place is p_1; p_1 in $O_t(t_1)$ and $|O_t(t_1)| > 1$, that is p_1 is one but not the only output place of t_1; and $I_p(p_1) = O_p(p_1) = \{t_1\}$, that is t_1 is p_1's only input and only output transition. A driven cycle of a time-driven system is available on Figure 12 on page 37. The dynamic result of the driving cycle construction is the firing of the transition t_1 precisely once every T_1 time units.

Using the basic concepts presented, the Petri net model of a time-driven system may be formed by adding places and transitions such that:

- each place has a fixed positive finite execution time (to model a task) or a zero execution time (to model a condition).
- the bipartite nature of the Petri net is preserved, that is arcs from places always go to transitions and arcs from transitions always go to places.
- to every place and transition added there exists a directed path from the transition t_1 in the driving cycle.
- eventually, all the paths terminate in transitions representing outputs of the system.

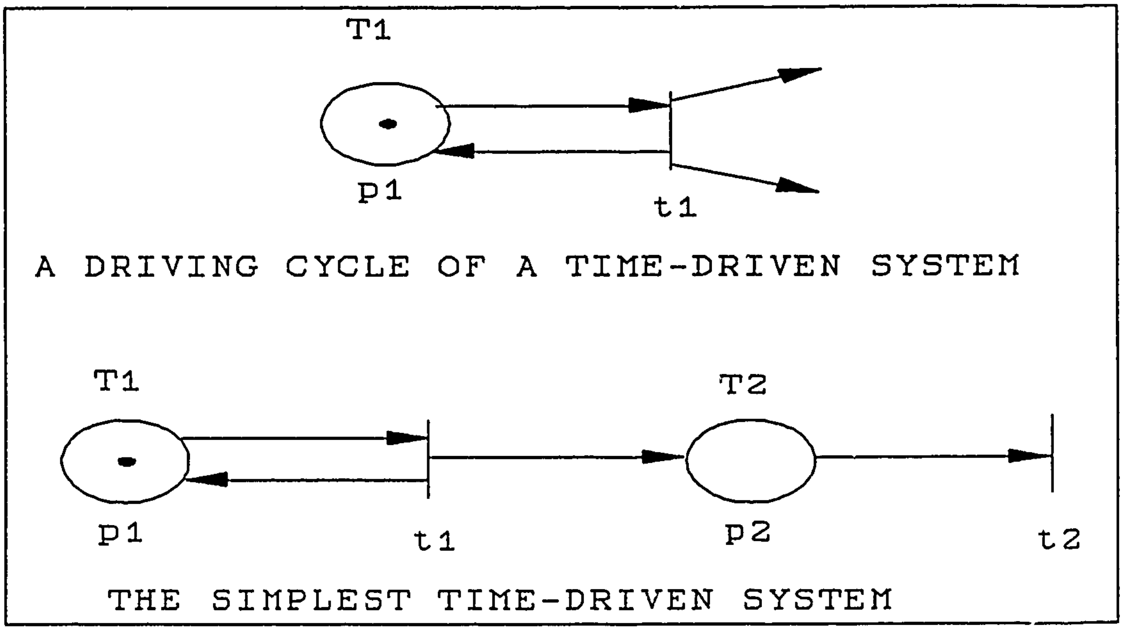

T1
p1
t1
A DRIVING CYCLE OF A TIME-DRIVEN SYSTEM
T1
T2
p1
t1
p2
t2
THE SIMPLEST TIME-DRIVEN SYSTEM

Figure 12 A Driven Cycle for a Time-Driven System

This procedure essentially "roots" the Petri net model in the driving cycle and ensures that the execution frequency of each process modeled by the Petri net is dependent on the firing frequency of the transition in the driving cycle.

The determination of the frequency at which a transition in the net fires relative to the firing of the transition in the driving cycle plays an important role in the analysis of the net construction, because the relative firing frequency is directly related to the interarrival time of consecutive tokens at a place. The two key concepts are the maximum relative firing frequency (MRFF) of the input transition of a place and the minimum token interarrival time (MTIAT). These are defined as follows:

- The MRFF of a transition is the number of times the transition fires for each firing of the driving cycle transition, assuming that all decisions lying between the driving cycle and the transition are made in favor of the path to the transition.
- The MTIAT of a place is the shortest possible time between the arrivals of any two consecutive tokens.

In many cases, the MRFF is found directly from the net consistency computations, it is only when a decision (multiple-output) place is encountered that additional information or assumptions are required. Decisions may be divided into two classes: predetermined and data-dependent. For determined decisions, it is known during the modeling process how often each output path is to be taken, and the decision as to which path is taken during execution is not based on data or another uncontrollable parameter. For data-dependent decisions, the modeler does not know precisely how often each path will be taken since this decision is dependent on data or some other dynamic parameter.

Predetermined decisions in time-driven systems may be used to permit a single driving cycle to be the timing basis for several processes operating at different basic timing rates. On the other hand, data-dependent decisions require that some assumption be made as to the frequency with each path will be taken. For most time-driven systems, it will generally be desired to evaluate the system under the worst case assumption. This requires considering the effect on transition firing frequencies when each output transition of the data-dependent decision is assumed to fire at the same frequency as the place's input transition, and gives raise to the MRFF definition.

Four analyzable subclasses of time-driven using Petri net have been defined in [Ref. LS87]. These subclasses are cited bellow:

- Asynchronous systems may be defined in terms of the cardinality of the sets of inputs and outputs of their places and transitions. For each place p_i and for each transition t_j (excluding the driving cycle transition) the following conditions hold : 1) $|I(p_i)| = |O_t(t_j)| = 1$, 2) $|O_p(p_i)| >= 1$. The Figure 13 on page 39 shows an example of an asynchronous time-driven system.
- Synchronized systems permit all of the constructions used in asynchronous systems, but also permit the use of synchronized parallel path constructions. A synchronized parallel path construction consists of a set T of transitions and a set P of path places p_p. T consists of one initial transition t_i, one final transition t_f, and a set T_p of zero or more path transitions t_p. P and T_p each consist of n (2 or more) disjoint

subsets such that P_i union with T_{pi} represents a path from t_i to t_f. Figure 14 , on page 40, shows an example of synchronized time-driven system.

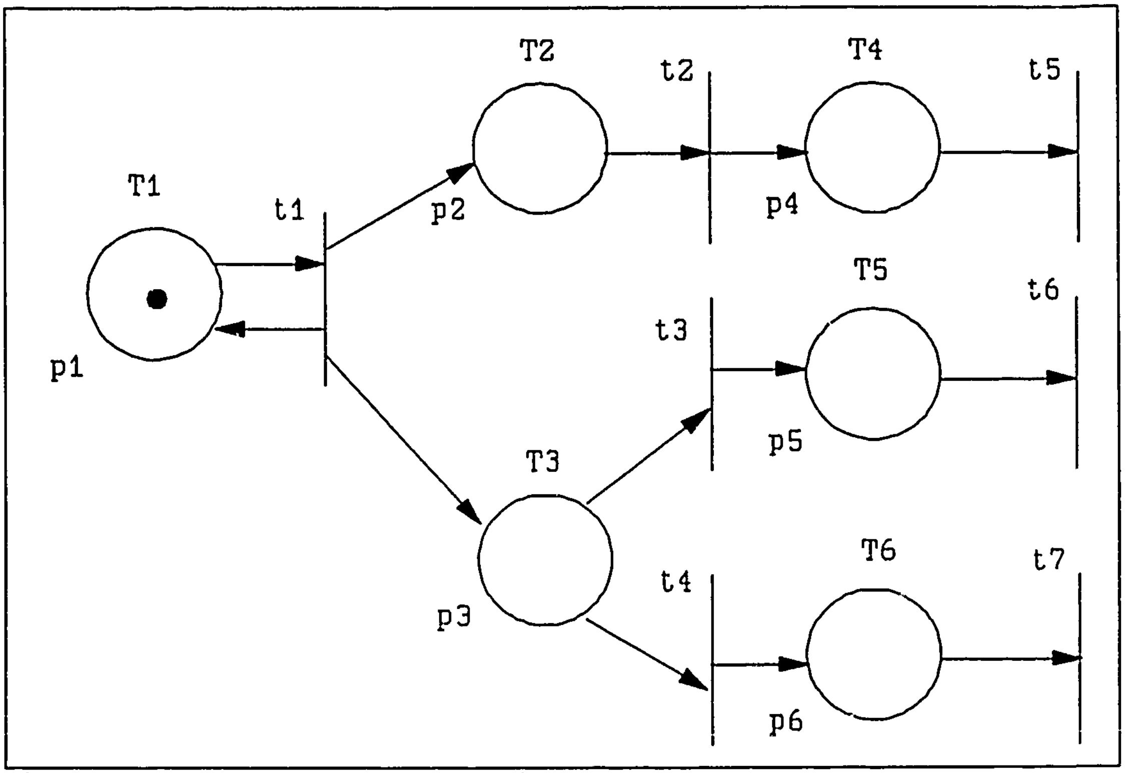


Figure 13 An Asynchronous Time-Driven System

- Independent cycle systems permit all of the constructions in synchronized systems, but also permit cycles to be formed by multiple inputs to transitions provided that all the cycles so formed are independent. The place outside the cycle with one input and one output which provides an input to the cycle will be called the entry place. An *independent cycle* consists of a set T of transitions and a set P of cyclic path places p_p. T consists of a cycle input transition t_i and a set T_p of zero or more cyclic path transition t_p. The union of T and P represents a cyclic path beginning and ending at t_i. The cyclic path place which is an input to t_i is marked initially with a single ready token so that t_i will fire immediately when the first token is ready at the entry place.

  - Shared resource systems provide a significant extension to independent cycle systems, since they allow cycles to overlap in such a way so as to permit the modeling of competition of shared resource. This is done by the addition of a shared construction. A *shared resource* construction is a set of n (2 or more) otherwise nonintersecting independent cycles, each of whose input transitions have a common firing frequency under all conditions, but which have been modified by replacing their final places with a common shared resource. A shared resource consists of a set T of zero or more resource path transitions t_p and a set P of places. P consists of an initial place p_i, a final place p_f (possibly the same as p_i) and zero or more resource path places p_p. A shared resource construction is shown is Figure 15 on page 41.

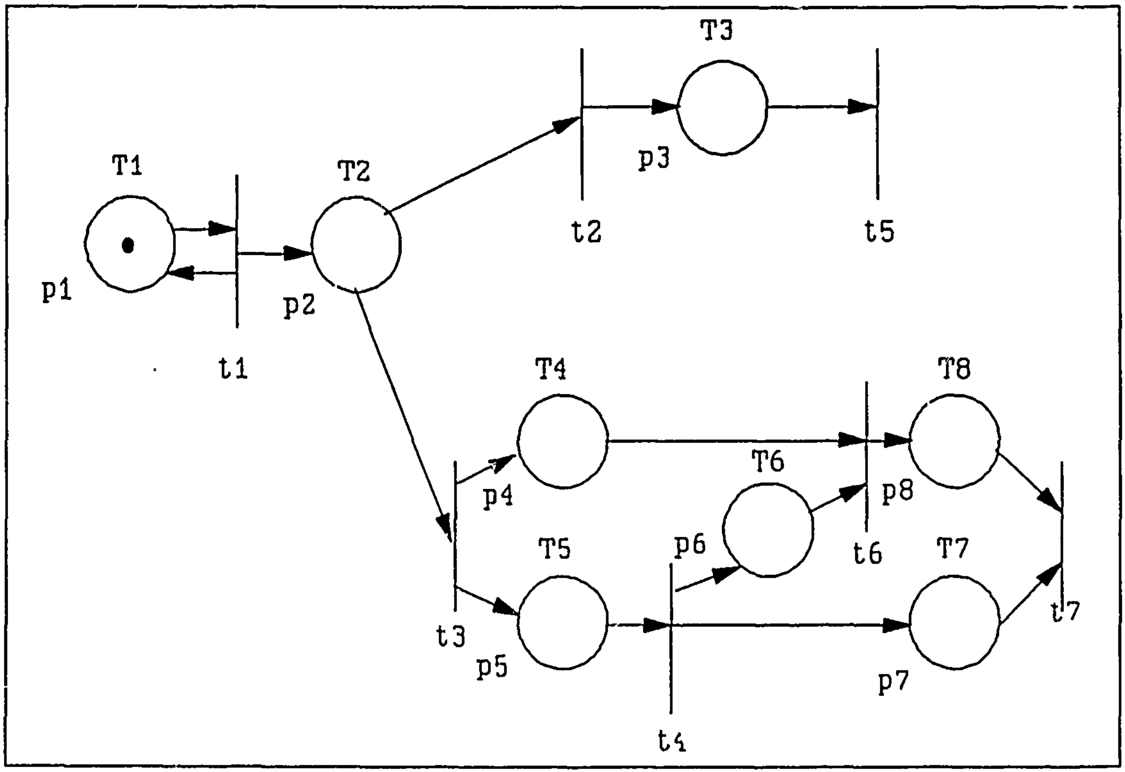


Figure 14 A Synchronous Time-Driven System

The construction of an analyzable timed Petri net model of a time-driven system consists of integrating the building blocks described above.

The following procedure may be used to construct a Petri net model of a time-driven system:

1) Construct the driving cycle;

2) As required by the system being modeled, add output places to the transition such that each place has:

   - a single input arc and,
   - either zero execution time (for a condition) or a finite positive execution time (for a task);

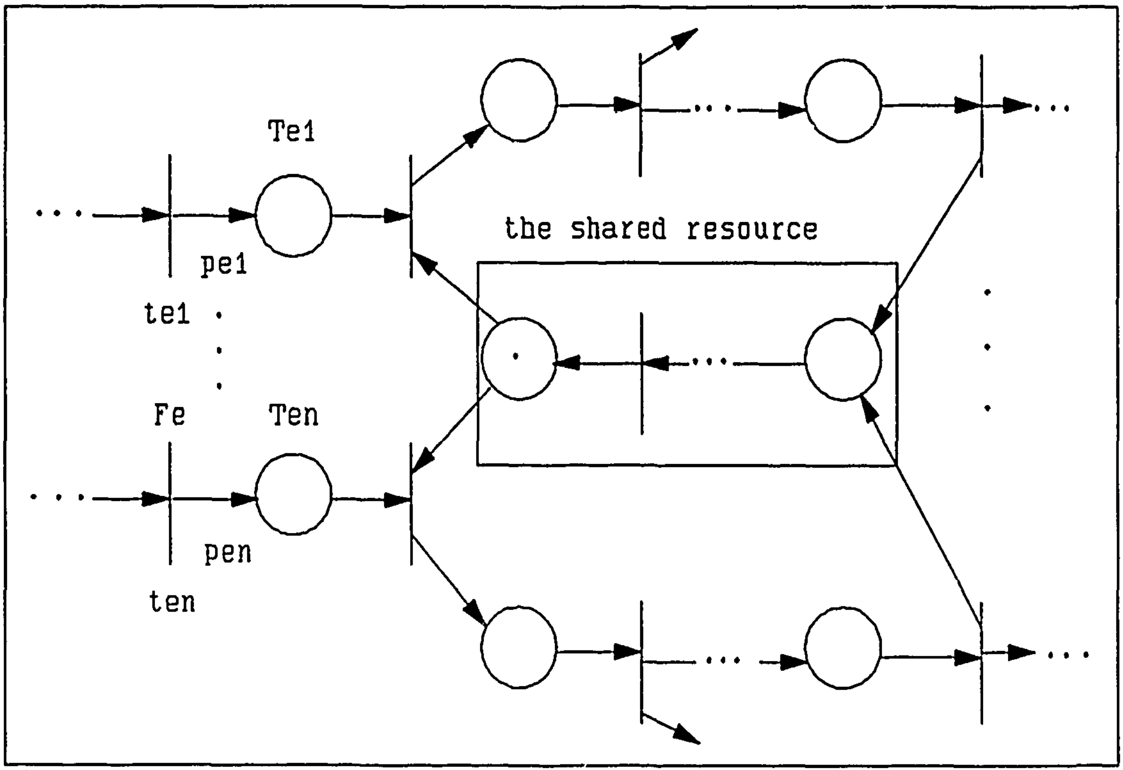


Figure 15 A General Shared Resource

3) As required, to each place as yet having no output, add one or more of the following net constructions as output:

- a single transition with exactly one input arc,
- a complete synchronized parallel path construction,
- a transition with multiple inputr which will complete a synchronized parallel path constxuction,
- an independent cycle,
- a cycle which forms part of a shared resource construction guaranteeing that all entry places have input transitions which will fire at the same frequency;

4) As required, to each transition as yet having no outputs (which is not an output of the net), add one or more output place (as in step 2);

5) Repeat steps 3) and 4) until the system has been completely modeled.

The explanation presented is this section is just a short review of the work done in [Ref. LS87], for more details and formal proof of all the criteria and constructors, as well analysis of the safeness of the entire Petri net the reference cited above is mandatory reading.

The benefits of this methodology are the following:

- Timing requirements may be stated formally and specifically, without the need to assign a time to each task individually.

Since the net constructions are well-defined, automated methods may be used to model a hard real-time system, because its possibility to assign different time-driven systems this model may be powerful to solve the case of scheduling problem in a multiprocessor environment. The basic concepts under this approach differ from the concepts of the graph model described in [Ref. Mok85a] and [Mok85b], and then this approach is not applicable to the solution of the static scheduler problem under the management of the CAPS system.

11. The Sequencing via Modular Decomposition

This approach assumes that the sequencing problem consists of a set of n tasks, wherein each task is described by the following characteristics:

- the ready time (a_i), the earliest point in time at which processing may begin on i,
- the processing time (d_i), the worst-case interval over which task i will occupy the processor,
- the deadline (b_i), the latest completion time for task i,
- all the tasks are nonpreemptable.

It is also assumed by this approach that there is a precedence constraint over the set of tasks that is possible to describe in terms of an acyclic directed graph.

The optimal sequencing via modular decomposition approach also assumes that exists a cost function that may be associated with the execution of each permutation of the tasks in the set.

The main idea in the theory of sequencing and scheduling is the method of adjacent pairwise task interchange. This method compares the costs of two sequences which differ only by interchanging a pair of adjacent jobs. In 1956, W. E. Smith defined a class of problems for which a total preference ordering of the tasks exists with the property that in any sequence, whenever two adjacent tasks are not in preference order, they may be interchanged with no resultant cost increase. [Ref. Smi56 : pp. 59-66]. A number of papers have generalized the adjacent pairwise interchange property to the adjacent sequence interchange property, whereby adjacent sequences of tasks are interchanged [Ref. Law78 : pp. 75-90, Ref. MS79 : pp. 215-224]. This generalization has resulted in efficient algorithms for precedence-constrained problems where the generality of the precedence oriented graph is restricted to some specific types of structures.[Ref. Sid81 : pp. 190-204]

With the recent development of efficient algorithms for locating modules in a precedence directed graph, a new class of sequencing algorithms has been promising [Ref. CS82 : pp. 214-228, Ref. BM83 : pp. 170-184, Ref MJ89 : pp. 1-19]. These algorithms obtain optimal sequences by finding optimal subsequences for progressively larger modules, until all the tasks are sequenced. To guarantee optimality of such algorithms, the cost function must satisfy the "job module property", which states that any optimal solution to a subproblem defined by a job module is consistent with at least one optimal solution for the entire problem. These results imply that precedence constraints built up iteratively from prime posets of width bounded by some fixed value can be solved in polynomial time.[Ref. MS87 : p. 22-31]

The basic algorithm used in this approach is the following:

1. The inputs are: the sequencing function, the precedence constraints, and the data about each task,
2. Find the composition tree T of the precedence constraints directed graph,

3. Find any node M in the composition tree, all of whose sons are presequenced,

4. Using dynamic programming, find an optimal sequence for the subproblem on M,

5. Replace M in T by the optimal sequence generated in 3. M is now presequenced by this sequence,

6. If all nodes of T are presequenced stop; the sequencing corresponding to the root of T is the optimal permutation. Otherwise go to 3.

A more detailed and deeper analysis of each aspect of this algorithm, as well its applicability to scheduling the tasks in a rapid prototyping system such the CAPS will be shown in the Chapter III during the theoretical development of the optimal static scheduling algorithm.

D. SUMMARY

The survey presents a sample of previous scheduling algorithms for hard real-time systems. Many of the algorithms discussed do not address the problem of how to schedule tasks that have precedence constraints. When there is a constraint on the earliest ready time usually an algorithm based in a tree branch and bound is used. The concept of a cost function to evaluate the schedule was shown in the minimize maximum tardiness with early start times scheduling algorithm. When precedence constraints were considered in the algorithms the solution adopted is to use some kind of graph representation (directed graph or Petri nets), and the notion of a base timeframe is used (harmonic block for the directed graph representation, and timing driven cycle for the Petri nets). None of the algorithms presented an optimal solution to the problem of scheduling hard real-time system with precedence constraints.

Petri nets seems to be good way to try to solve the problem, but, as stated in the reference, there is a lot of work to be done before the theoretical basis is well established. Based on Petri nets is possible to model a hard real-time system using an automated tool (similar to CAPS), but this tool is not yet available. As stated in the section II.C.9 this tool will allows the development of automated system, similar to CAPS, designed to solve the scheduling problem of hard real-time systems.

The approach that will be followed in this thesis is to refine and extend the ideas developed in the harmonic block with precedence constraints scheduling algorithm (in order to define a timeframe). Instead of using a topological sort of the operators we will consider all the instances of the operators that occur during the timeframe. The new graph obtained from the two concepts above will be analyzed with the new tools available from the recent developments in the analysis of directed acyclic graphs (theories about sequencing via modular decomposition and enumerative procedures).

The evaluation criterion for the work will be a modified version of the tardiness cost function defined in the minimize maximum tardiness with early start times scheduling algorithm.

III. DESIGN OF AN OPTIMAL STATIC SCHEDULING ALGORITHM

In this chapter we develop two approaches for the optimal scheduling problem for a single processor. In order to define the level of difficulty of the problem in question we introduce the concept of non-polynomial problem. A detailed definition of a task , its parameters and the correlation with the operator is explained in detail. The graph of constraints, which combines the precedence constraints of the tasks with some time information is described in detail. The cost functions applicable are introduced and analyzed. The algorithms for the two approaches are described in detail.

A. NON-POLYNOMIAL PROBLEMS

For many years many researchers have been trying to find efficient algorithms for solving various combinatorial problems, with only partial success. Some of these problems are: the simplification of Boolean functions, scheduling problems, the traveling salesman problem, certain flow problems, covering problems, placement of components problems, minimum coloration graphs, winning strategies for combinatorial games.

Since all the problems we consider are solvable, in the sense that there is an algorithm for their solution (in finite time), we need a criterion for deciding whether an algorithm is efficient[4]. The length of the data describing the instance is called the input length. This length depends on the format chosen to represent the data (for graphs we can use an adjacency matrix, or incidence lists, etc.). An algorithm is efficient if there exists a polynomial p(n) such that an instance whose input length is n takes at most p(n) elementary computational steps to solve. That is, we accept an algorithm as efficient only if it is of polynomial time complexity. This is a crude criterion, since it says nothing about the degree or coefficients of the polynomial. In practical applications where n is small the degree of the polynomial and the size of the coefficient are significant.

Assume we have two algorithms for a solution of a certain problem. Algorithm P_1 is of complexity n^2 and algorithm P_2 is of complexity 2^n. Let n_0 be the longest instance (n_0 is its input length) which can be solved by algorithm P_2, using a given computer A. Now if we have a computer B ten times faster, the largest

[4] A good discussion and analysis of algorithms is available in Aho, D.L., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Menlo Park, California, 1974.

instance n we can handle must satisfies the equation $n = n_0 + \log_2 10$, this means that the new input length is $n_0 + 4$, this is not a very dramatic improvement. However, if n_0 is the largest instance we could handle, by A, using algorithm P_1, then now we can handle, by B, instances of length up to n where $n^2 = 10\ n^2_0$. This means that we would be able to handle input length with more than three times the original input length.

The class of problems in NP can be defined as a class of language recognition problems solvable by a nondeterministic Turing machine in a number of steps bounded by a polynomial in the size of the problem input length[5]. Another important aspect, in order to analyze the complexity of an algorithm, is the concept of polynomial reducibility. Consider two problems P_1 and P_2: we say that P_1 is polynomially reducible to P_2 if for any instance of P_1 an instance of P_2 can be constructed in a polynomial-bounded number of steps such that solving the instance of P_2 will solve the instance of P_1 as well. Thus, P_1 can be informally considered to be a special case of P_2. If P_1 polynomially reduces to P_2 and also P_2 polynomially reduces to P_1 then the two problems can be considered equivalent from the point of view of computational complexity. The literature available shows that all the NP problems are polynomially reducible to the "satisfiability problem".[Ref. Coo71 : pp. 151-158] Based on this the class of NP-complete problems can be defined as those problems in NP to which the satisfiability problem polynomially reduces, or equivalently a problem from NP is NP-complete if all problems from NP can be polynomially reduced to it. Since the satisfiability problem is a decision type problem, requiring a yes or no answer for the question whether a given Boolean variables can assume the value true, it is customary to formulate NP-complete problems as decision problems requiring yes or no answer. For combinatorial optimization where the solution is in the form of an optimal solution, the terminology NP-hard is used often, if the problem formulated as a decision problem is NP-complete. It has been proved that some known NP-complete problems can be reduced to certain scheduling problems.[Ref. GJ78]

Since many practical problems are "intractable" in this sense, hope for producing algorithms which would find optimal solutions in a reasonable amount of time had to be abandoned and instead attention was directed to the development and analysis of heuristic algorithms. It was soon realized that the very

[5] A rigorous mathematical definition of Turing machine is described in Even,S., *Graph Algorithms*, Computer Science Press, 1979.

pessimistic worst-case analysis included in the NP-complete results did not accurately reflect the success which heuristic algorithms were achieving on real-world combinatorial algorithms.

The search for a theoretical answer to this question leads to three main approaches. The first allows approximations to be made to the optimal solution. In a very few cases a constant bound on the ratio of the approximation to the optimal may be proven however in most cases such a bound is not known. Furthermore even if such a bound is known, in practice the observed behavior of the algorithm is often much better than the bound. The second approach substitutes either expected case or average case analysis for worst-case analysis. Typically this type of research involves the probabilistic analysis of a particular heuristic algorithm. Often this is accomplished by analyzing the algorithm's behavior on some random input. One shortcoming of this approach is that usually the graphs encountered in practice have some structure which violates the assumption that all edges have the same independent probability of being present. For this reason the algorithm's performance in a real-world environment often is not as good as predicted by the optimistic expected case or average case analysis.

The third approach, and the one we take in this thesis, maintains the worst-case analysis and restricts the class of input to be considered. The hope here of course is that the intractable problem will be solvable in a reasonable amount of time, if possible in polynomial time based in some characteristic of the input data [Ref. CPS85 : pp. 926-934]. The restrictions that are applied to the class of input are discussed in the further sections. Examples of families of graphs which have received this type of study, with some degree of success, include comparability graphs, permutation graphs, interval graphs and planar graphs.

B. OPERATORS AND TASKS

The PSDL language is based on a computation model which treats software systems as networks of operators communicating via data streams. The computational model is an augmented directed graph G = (V,E,T(v),C(v)), where V is the set of vertices, E is the set of edges, T(v) is the set of timing constraints for each vertex v, and C(v) is the set of control constraints for each vertex v.

All PSDL operators are state machines. Some PSDL operators are functions, i.e. machines with only one state. When a operator fires, it reads one data value from each of its inputs streams, undergoes a state transition, and writes at most one data value into each of its output streams. The output values can depend only on the current set of input values and the current state of the operator. St ıte transitions and input/output operations on data streams can occur only when the associated operator fires. The firing of an operator is controlled by the associated timing and control constraints. Operators can be triggered by the arrival of a set of input data values or by a periodic temporal event.[Ref. Luq89 : p. 77-8]

The operators in PSDL may be atomic or composite. The atomic operator is defined as the basic indivisible unit of work to be executed, and the composite operator is defined as being an operator that can be decomposed into atomic operators. Two possibilities of decomposition of a composite operators exist: linear decomposition and network-like decomposition.[Ref. Jan88 : pp. 34-35, Ref. Mar88 : pp. 55-56] Figure 16, on page 50, illustrates the two possible decompositions.

The first restriction that we impose on the scheduling problem (comming from the PSDL source file) is that all the operators must be atomic. This means that all the operators in the scheduling problem are indecomposable, or already had been decomposed into their atomic components.

Any PSDL operator can have timing constraints associated with it. An operator is time-critical if it has at least one timing constraint associated with it, and is non time-critical otherwise. There are several different kinds of timing constraints, which can be classified into those that apply to all time-critical operators, those that apply only to operators triggered by periodic temporal events, and those that apply only to operators triggered by the arrival of new data.

Every time-critical operator must have a maximum execution time (MET). The MET of an operator is an upper bound on the length of the execution interval (EI) for the operator. All the actions that may be required to fire an operator once must fit into the execution interval. These actions are listed bellow.

- Reading values from input data streams,
- Evaluating triggering conditions,
- Calculating output values,
- Evaluating output guards,
- Writing values into outputs streams.

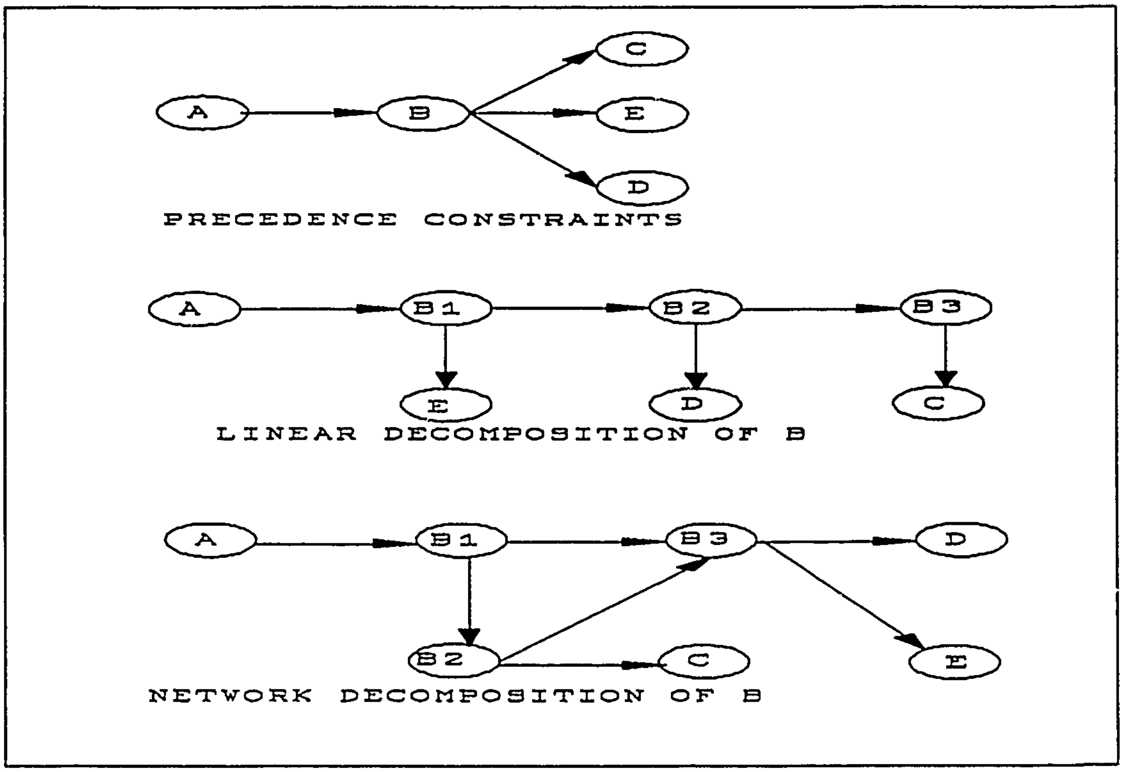


Figure 16 Linear vs. Network Decomposition

The execution interval for an operator does not include scheduling delays. A scheduling delay is the time between the writing of a value into a data stream by a producer and the reading of that value by the consumer operator.

Operators triggered by temporal events are periodic in PSDL. Every periodic operator must have a period (PERIOD) and may have a deadline (FINISH_WITHIN). These two time constraints partially determine the set of scheduling intervals (SI) for each operator. Each periodic operator must be fired exactly once in each scheduling interval, and must complete execution before the end of the scheduling interval. The period is the length of time between the start of any scheduling interval and the start of the next scheduling interval. The deadline is the length of each scheduling interval. The relation between the timing constraints, scheduling intervals, and execution intervals for a periodic operator is illustrated in Figure 17, on page 71 [Ref. Luq89]. The execution intervals and scheduling intervals in the diagram are indexed by integers in order of their occurrence. Thus SI[n] denotes the n^{th} scheduling interval for the operator and EI[n] denotes the n^{th} execution interval for the operator.

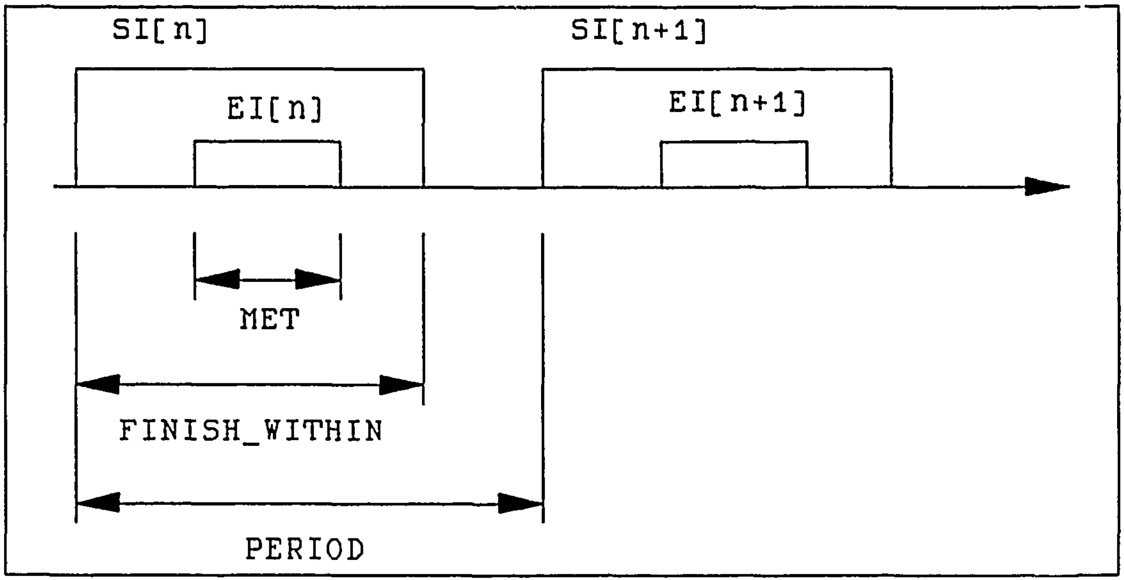


Figure 17 Timing Constraints for a Periodic Operator

Operators triggered by the arrival of new data values are sporadic. Timing constraints for sporadic operators are optional. Sporadic operators with timing constraints must have both a maximum response time (MRT) and a minimum calling period (MCP) in addition to an MET. The MRT is an upper bound on the response time, while the MCP is a lower bound on the calling period. The relation between these quantities is illustrated in the Figure 18, on page 52 [Ref. Luq89]. SI[n] denotes the n^{th} scheduling interval for the consumer operator, which is sporadic and time-critical. CEI[n] denotes the n^{th} execution interval for the consumer operator, and PEI[n] denotes the n^{th} execution interval for the producer operator, which is assumed here to be included in the definition of the scheduling problem. The response time associated with a consumer operator is measured from the end of the execution interval for the producer operator of the triggering data value to the end of the execution interval for the consumer operator of the triggering data value.

Unlike the MET, the MRT includes a scheduling delay. The MRT gives the length of the scheduling interval for a particular triggering data instance. The calling period of an operator is the length of time between the end of the execution interval for the producer of the triggering data value and the end of the execution interval for the producer of the next triggering data value. The calling period must not be less than the MCP. The MCP of an operator constrains the behavior of the producers of the triggering data values rather than constraining the behavior of the operator itself. An MCP constraint is needed to allow the realization of a maximum response time constraint with a fixed amount of computational resources, via a limit on the frequency with which new data can arrive.[Ref. Luq89 : pp. 10-13]

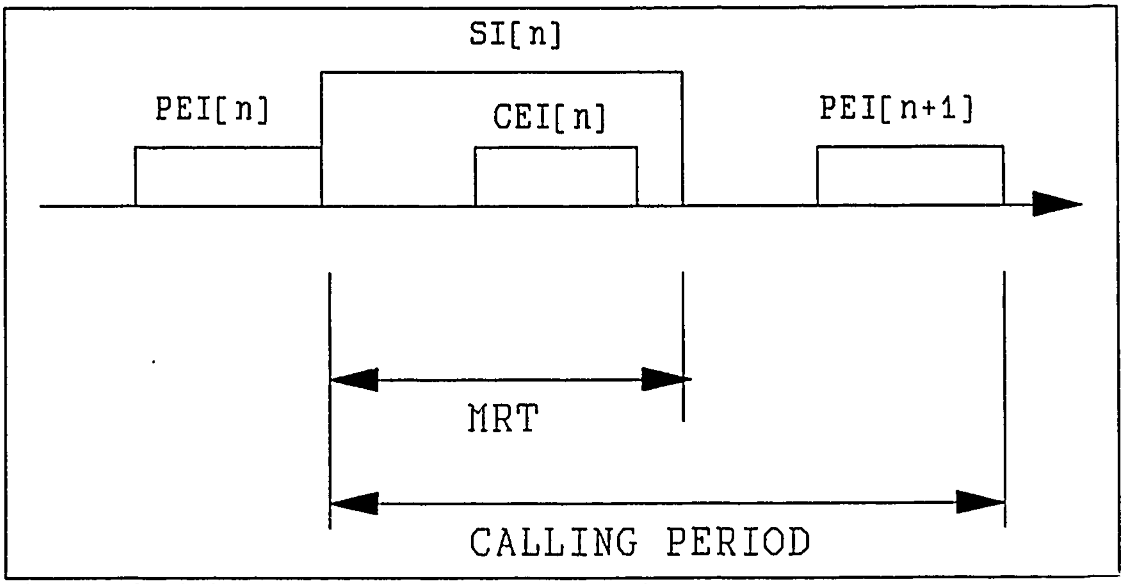


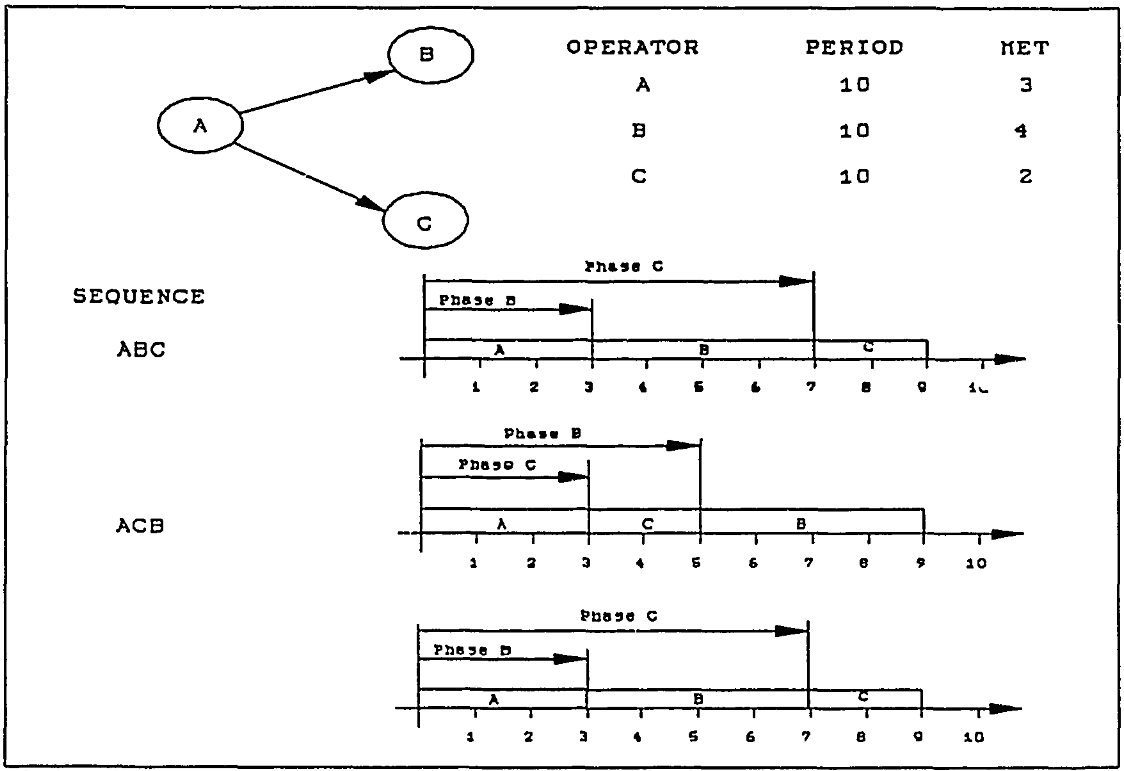
Figure 18 Timing Constraints for a Sporadic Operator

The second restriction that we impose on the scheduling problem is that all the operators must be periodic. In order to handle the sporadic operators they are converted into their correspondent periodic equivalent operators. The conversion may be done using the procedure described in [Ref. Mar88 : p. 12], where the period of the equivalent periodic operator is defined as PERIOD = minimum(MCP, MRT-MET), an equivalent periodic operator derived in this manner has a deadline equal to the maximum execution time.

The third restriction on the scheduling problem, with respect to the operators, ıs that only the time-critical operators are analyzed in order to obtain the optimal static schedule. The non time-critical operators are handled by the Dynamic Scheduler.

Since each periodic operator fires more than once during the execution of the problem, we define each firing of an operator to be a separate task to be scheduled.

Another characteristic of an operator is its phase, which is defined as the delay between the reference time zero and the starting time of the first scheduling interval for this operator. The phase is a function of the operator and the permutation of operators chosen by the scheduler. This fact is illustrated in Figure 19 on page 53.



OPERATOR	PERIOD	MET
A	10	3
B	10	4
C	10	2

Figure 19 Possible Phases of an Operator

C. GRAPH OF CONSTRAINTS

In the current version of the Static Scheduler being utilized by CAPS, the graph of precedence constraints and the periodicity of each operator are analyzed independently. In order to analyze the set of constraints using the tools available for enumeration or for modular decomposition it is necessary to have all the tasks defined, as well as all the interconnections among them.

The precedence constraints are defined by communications among the operators that compose the system being developed. PSDL operators communicate by means of named data streams. PSDL data streams can carry either normal data or tokens representing exceptions. All of the data values carried by a stream must be instances of a specified abstract data type associated with the stream.

There are two different kinds of data streams in PSDL, dataflow streams and sampled streams. Dataflow streams are used in applications where the values in the stream must not be lost or replicated and the firing rates of the producer and consumer are the same, while sampled streams are used in applications where a value must be available at all the times and can be replicated without affecting their meaning. Each data stream represents a directed data path between two operators, and the producer is defined as the predecessor of the consumer (and conversely the consumer is the successor of the producer).

We expand the augmented graph defined in section B as: V is the set of all the operators, E is the set of all the dataflow paths, T(v) is the set of timing constraints of each operator and contains the following data fields:

1. T(i).PERIOD,
2. T(i).TIME_ALLOWED, equivalent to FINISH_WITHIN or MRT,
3. T(i).NUMBER_OF_INSTANCES[6],
4. T(i).FIRST_INSTANCE[6], and
5. T(i).PHASE[6]

The set V may be represented as a vector of strings or integers (OPERATOR_ID), where those values represent the operator identifier; the set of precedence constraints (dataflow paths) E may be represented as a adjacency matrix, where the element E(i,j), i and j are natural numbers, is 1 if the OPERATOR_ID(i) is predecessor of OPERATOR_ID(j), and zero otherwise; the set of timing constraints T can be represented as a vector of records where each record contains the timing elements defined above, the vector V and the vector of T(v) records have corresponding components (the operator V(j) has the timing constraints T(j)).

In order to simplify further manipulations on the data two dummy operators are included in the set V: the dummy operator V(1), and the dummy operator V(n+2), where n is the number of operators in the original set V furnished by the user. The row 1 and n+2, and the columns 1 and n+2 of the adjacency matrix are defined as having only the 0 (zero) value, n+2 is denoted by N.

At this step the following constraints may be checked for the set of timing constraints (except for the two dummy operators):

1. T(i).MET <= T(i).TIME_ALLOWED,
2. T(ı).MET <= T(i).PERIOD, and
3. T(i).TIME_ALLOWED <= T(i).PERIOD.

We assume that only well formed sets are submitted to the optimal scheduling algorithm, and that constraints violations lead to an error message and early termination of the scheduling process.

[6] These parameters of an operrator are not defined by the user, they are computed by the scheduling algorithm during the analysis of the problem.

In order to obtain the graph of constraints we need to define a timeframe. The approach that we selected is to define a harmonic block as in [Ref O'He88 : pp. 34-41]. This harmonic block is repeated indefinitely and ensures that all the time-critical operators are performed within their timing constraints (if a feasible solution to the proposed problem exists). This means that our graph of constraints represents one instance of the harmonic block.

Three steps are necessary to obtain the graph of constraints. The first is to find the length (in time) of the harmonic block. After this length is defined we need to define which tasks must be scheduled as well as the precedence constraints among these tasks. The third step is to order all the tasks.

1. Length of the Harmonic Block

The length of the harmonic block is simply the least common multiple (LCM) of all the operators that belongs to the set in analysis. Z is defined as the LCM of (X,Y) if and only if Z mod X = 0 and Z mod Y = 0 and (W mod X = 0 and W mod Y = 0) implies that Z <= W. The LCM is computed by taking two periods at a time, multiplying them together, and then dividing this result by the greatest common divisor (GCD)of the two periods. This result is then multiplied together with the next period and divide by their GCD until all operators in the set have been processed. The result of this operation on the last pair in the set is the LCM of all operators in the set.

The algorithms for the GCD and LCM are presented below:

a. Algorithm for GCD

```
define gcd(a,);
  if b > a then define_gcd(b,a)
  else if a mod b = 0 then b
   else define_gcd(b, a mod b)
  end if;
end define_gcd.
```

Example:

Operator	Period
1	-
2	6
3	4
4	14
5	-

The application of the algorithm gives: GCD = 2.

b. Algorithm for LCM

```
define_lcm;
  N := size(V);
  LCM := 1;
  for I in 2 .. N - 1 loop
     P := T(I).PERIOD;
     LCM := LCM * P / gcd(LCM,P);
  end loop;
end define_lcm;
```

Example: Using the same set of data as before we obtain the following result : LCM = 84.

2. Tasks of the Graph of Constraints

The tasks are the instances of each operator that must be executed inside the timeframe, the number of tasks for each operator is obtained by dividing the length of the harmonic block by the period of the operator. The result of this operation is stored in the timing constraints record T(v) of each operator. After the evaluation of the number of tasks for each operator it is possible to check another input constraint. One condition necessary but not sufficient condition for the set to be feasible is that the sum of all the execution times of the tasks must be less than or equal to the length of the harmonic block.

The generation of the graph of constraints for the tasks is done in two steps. During the first step we produce a chain for each operator, and in the second step we use the precedence constraints among the operators in order to generate the precedence constraints among the tasks.

a. Algorithm for Number of Tasks

```
define_number_of_tasks;
  N := size(V);
  T(1).NUMBER_OF_TASKS := 1;
  T(N).NUMBER_OF_TASKS := 1;
  for I in 2 .. N - 1 loop
     T(I).NUMBER_OF_TASKS := LENGTH_HARMONIC_BLOCK /  T(I).PERIOD;
  end loop;
end define_number_of_tasks.
```

Example: Using the same set of data as before.

```
T(2).NUMBER_OF_TASKS = 14
T(3).NUMBER_OF_TASKS = 21
T(4).NUMBER_OF_TASKS = 6
```

3. Precedence Constraints of the Tasks

Before we generate the graph of constraints some definitions are necessary. In this thesis a task *i* is said to have precedence over task *j*, denoted by *i* -> *j*, if task *i* must occur before task *j* in every feasible permutation, in other words *i* is the predecessor of *j*.

A partially ordered set (poset) is called a chain if exactly one permutation is feasible (if the problem has no precedence constraints then the correspondent poset is called an antichain, but this case is not applicable to us). As we can see, by the definition above, each set of tasks corresponding to the same operator is a chain, because we cannot execute the second instance of the operator unless we have already executed the first instance and so on.

We generate the precedence relations between the tasks (graph of constraints) the precedence relations between the operators via the following rule:

Suppose O_1, O_2 are operators and T_1, T_2 are corresponding tasks with instance numbers I_1, I_2 and periods P_1, P_2.
TASK(T_1).OPERATOR_NUMBER = O_1,
TASK(T_2).OPERATOR_NUMBER = O_2,
TASK(T_1).INSTANCE_NUMBER = I_1,
TASK(T_2).INSTANCE_NUMBER = I_2,
T(O_1).PERIOD = P_1,
T(O_2).PERIOD = P_2, then the task precedence relations are defined by the following:

If E(O_1,O_2) = 1 and $P_1 * I_1 = P_2$ then E_TASK(T_1,T_2) = 1, and otherwise E_TASK(T_1,T_2) = 0.

All the first instance of each operator i is precceded by the dummy operator 1 if and only if there is no other task that precceds i.

All the last instance of each opertor i precceds the dummy operator N if and only if they does not precced another task.

The data structure selected for each task is a record with the following data fields:

1. TASK(i).OPERATOR_NUMBER which represents the number of the corresponding operator in the vector V;

2. TASK(i).INSTANCE_NUMBER which represents the position of the corresponding operator in the chain, where the first instance has INSTANCE_NUMBER = 0;

The set of all the records representing the tasks is assumed to a be vector named TASK with length (TASK_LENGTH) defined by the sum of all the T(i).NUMBER_OF_INSTANCES.

The adjacency matrix for the tasks is a TASK_LENGTH x TASK_LENGTH square matrix. The principal diagonal of this matrix is equal to zero because no edges from task *i* to task *i* are allowed. The elements TASK(1) and TASK(TASK_LENGTH) (they are the only instances of the dummy operators V(1) and V(N), respectively) are dummy tasks used in the construction of the graph of constraints and the labeling process to be explained further.

a. *Algorithm for Generate Chains of Tasks*

```
generate_chains_of_tasks;
 [E_TASK] := [0];
 T := 2;
 N := size(V);
 for OP in 2 .. N - 1 loop
      LAST := T(OP).NUMBER_OF_TASKS - 1;
       for I in 0 .. LAST loop
           TASK(T).OPERATOR_NUMBER := OP;
           TASK(T).INSTANCE_NUMBER := I;
           if I > 0 then
                 E_TASK(T-1,T) := 1
           else
                 E_TASK(1,T) := 1;
           end if;
           if I = LAST then
                 E_TASK(T,TASK_LENGTH) := 1
           end if;
           T = T + 1
       end loop;
  end loop;
end generate_chains_of_tasks.
```

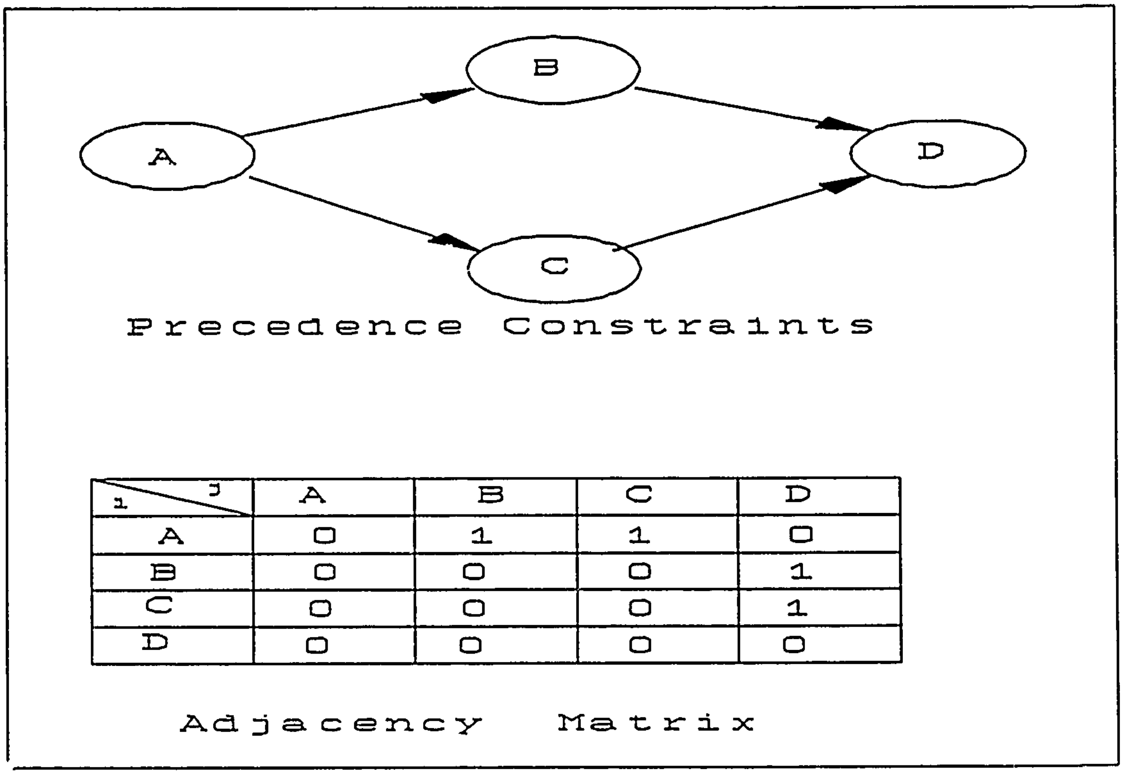
Example: A application of the algorithm above is illustrated on the Figure 21, on page 60. This example is based on the set defined on Figure 20, on page 59.

The data available for the set V and T is the following:

i	V(i)	T(i).PERIOD	T(i).NUMBER_OF_TASKS
1	dummy	-	1

2	A	10	3
3	B	15	2
4	C	5	6
5	dummy	-	1

and LCM = 30



Precedence Constraints

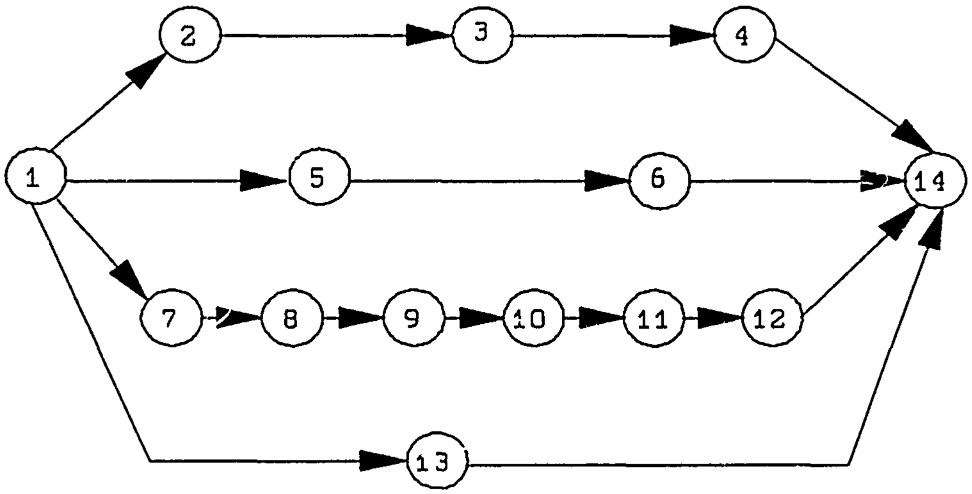
i \ j	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	0	0	0	1
D	0	0	0	0

Adjacency Matrix

Figure 20 Precedence Constraints

b. Algorithm for Interconnecting Chains of Tasks

```
interconnect_chains;
 for I in 2 .. TASK_LENGTH - 1 loop
     OP := TASK(I).OPERATOR_NUMBER;
     INSTANCE := TASK(I).INSTANCE_NUMBER;
     P1 := T(OP).PERIOD;
     for J in 2 .. TASK_LENGTH - 1
         OP2 := TASK(J).OPERATOR_NUMBER;
```



GRAPH OF CONSTRAINTS

TASK	OP	INSTANCE
1	-	-
2	1	0
3	1	1
4	1	2
5	2	0
6	2	1
7	3	0
8	3	1
9	3	2
10	3	3
11	3	4
12	3	5
13	4	0
14	-	-

TASKS DESCRIPTION

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	1	0	0	1	0	1	0	0	0	0	0	1	0
2	0	0	1	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	1
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	1
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 21 Chains of Tasks

```
         P2 := T(OP2).PERIOD;
         INSTANCE2 := TASK(J).INSTANCE_NUMBER;
         if OP <> OP2 then
             if P1*INSTANCE = P2*INSTANCE2 then
                 if E(OP,OP2) = 1 then
                     E_TASK(I,J) := 1;
                 end if;
         end if;
     end loop;
 end loop;
end interconnect_chains.
```

Example of this algorithm is shown in Figure 22, on page 62. The number outside the nodes represents the earliest start of the associated task.

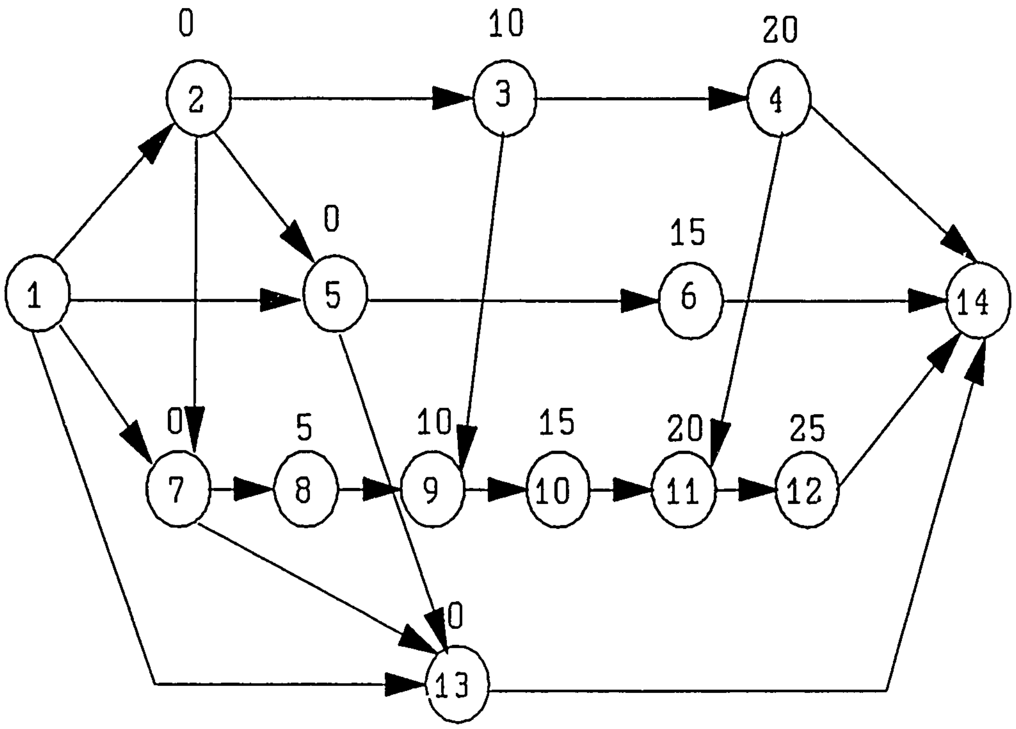
4. Ordering the Tasks of the Graph of Constraints

The graph of constraints obtained in the subsection 3 has all the data necessary in order to be utilized by the job modular decomposition but not for the enumeration techniques.

The enumeration techniques requires that if *i* is predecessor of *j* (E_TASK(i,j) = 1), then the integers associated with them must obey the relation n(i) < n(j). To ensure that the graph of constraints obeys this relation we must renumber the tasks. This is done by applying a topological sort to the graph of constraints, see [Ref. O'He88].

a. Algorithm to Reorder the Tasks

```
reorder_tasks;
 NEW_ORDER := [0];
 NEW_ORDER(1) := 1;
 NEXT_LEVEL := { 1 };
```



GRAPH OF CONSTRAINTS

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	1	0	0	1	0	1	0	0	0	0	0	1	0
2	0	0	1	0	1	0	1	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	1	0	0	1
5	0	0	0	0	0	1	0	0	0	0	0	0	1	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0	1	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	1
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ADJACENCY MATRIX

Figure 22 Graph of Constraints

```
  NEW_NUMBER := 2;
  while GRAPH_OF_CONSTRAINTS <> {} loop
      remove all nodes of NEXT_LEVEL and all the edges originated in them from the GRAPH_OF_CONSTRAINTS;
      generate set NEXT_LEVEL with all nodes without any incomming edges;
      NEXT_LEVEL1 := NEXT_LEVEL;
      while NEXT_LEVEL1 <> {} loop
          pickup one element from NEXT_LEVEL1 it is CURRENT_NODE;
          set NEW_ORDER(CURRENT_NODE) := NEW_NUMBER;
          remove CURRENT_NODE from NEXT_LEVEL1;
          NEW_NUMBER := NEW_NUMBER + 1;
      end loop;
   end loop;
   update TASK and E_TASK;
end reorder_tasks.
```

Figure 23, on page 64, illustrates an example of the application of the algorithm described above.

5. Description of the Steps to Obtain the Graph of Constraints

The graph of constraints is completely defined and evaluated using the algorithms described in the former subsections. In order to generate a DFD of the global algorithm to generate the graph of constraints the following naming is assumed:

- Evaluation of the GCD of the operators: define_gcd,
- Evaluation of the LCM of the operators: define_lcm,
- Evaluation of the number of tasks in the graph of constraints: define_number_of_tasks,

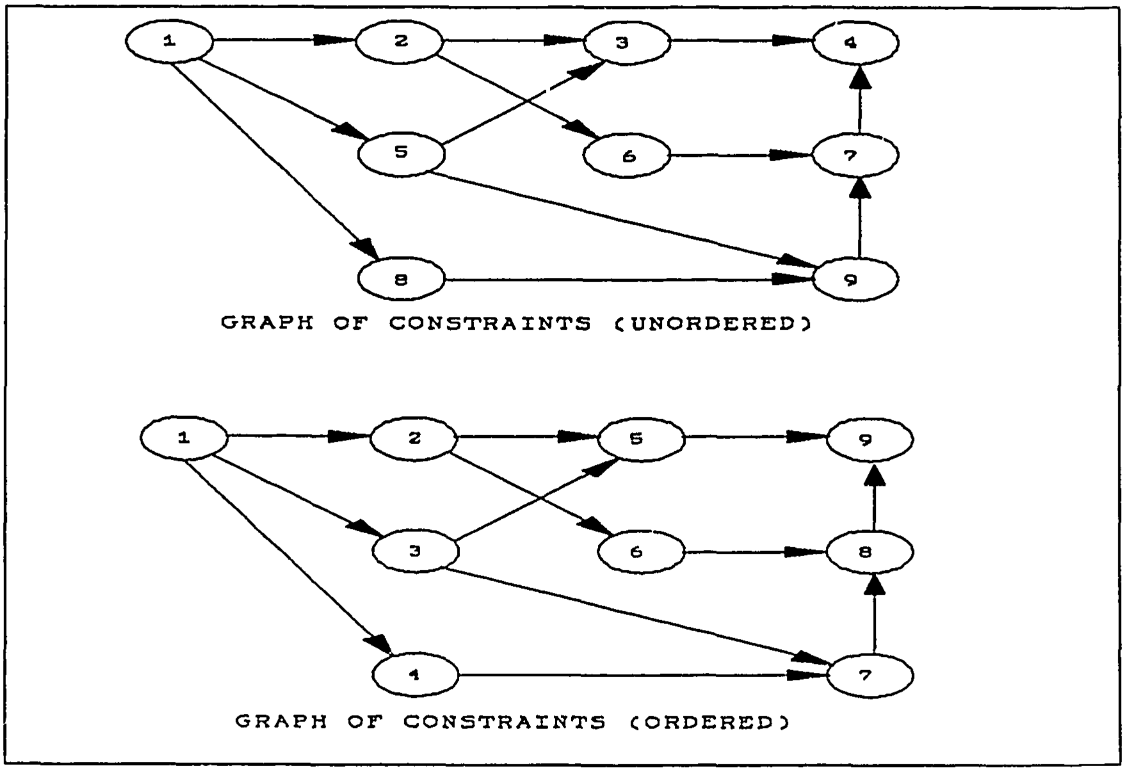

1
2
3
4
5
6
7
8
9
GRAPH OF CONSTRAINTS (UNORDERED)
1
2
5
9
3
6
8
4
7
GRAPH OF CONSTRAINTS (ORDERED)

Figure 23 Graph of Constraints Ordered

- Generation of chains of tasks: generate_chains_of_tasks,
- Interconnection of the chains: interconnect_chains,
- Reorder the graph of constraints: reorder_tasks,

The DFD of the GRAPH_CONSTRAINTS is illustrated in the Figure 24, on page 65.

C. COST FUNCTIONS

The performance objective of meeting task deadlines is one of the scheduling criterias most frequently encountered. While meeting deadlines is only a qualitative goal, it usually implies that time-dependent penalties are assessed on late jobs but no benefits derive from completing tasks early.

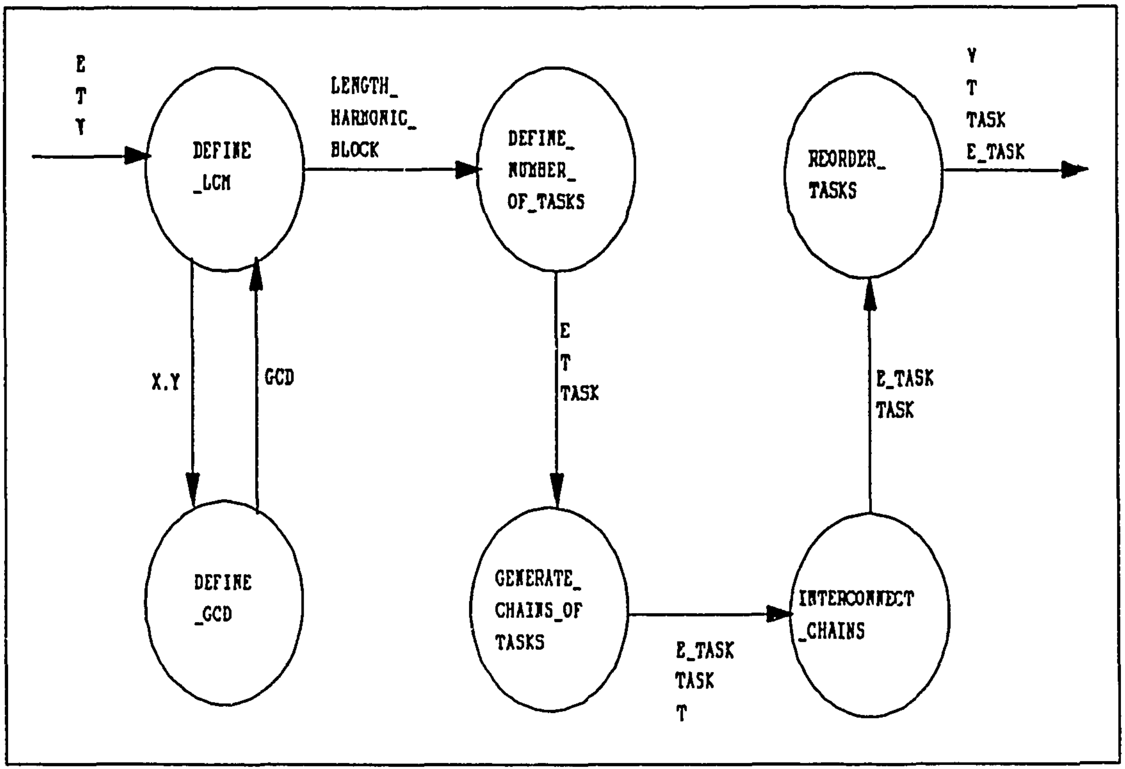


Figure 24 1st Level DFD Graph of Constraints

1. Preliminary Definitions

Consider n tasks to be sequenced on a single processor, where each task has the following attributes:

- MET_i : task i requires MET_i time units of processing,
- $PERIOD_i$: period of the base operator for the task i,
- $PHASE_i$: phase of the base operator for the task i,
- $INSTANCE_i$: instance of the t ask i,
- $EARLIEST\_START_i$: earliest start possible for the task i,
- $TIME\_ALLOWED_i$: maximum time allowed to finish the task i after the earliest start,
- $DEADLINE_i$: maximum completion time allowed for the task i,
- $TARDINESS_i$: the amount of time by which i missed its deadline,
- $COMPLETION_i$: time when the task i is finished.

Another concept needed for the analysis and evaluation of the cost functions is a sequence of tasks.

A sequence s consisting of *k* tasks is a function from {1,2,...,k} to the set TASK, defined in section C, and is represented by (s(1),s(2),...,s(k)), where s(i) is the ith task in the sequence s.

In this thesis the cost function assigns a integer value (or cost) to each sequence. The scheduling problem on a set TASK of tasks with cost function *f* is to find a permutation of TASK contained in a predefined set of feasible permutations *F* that minimizes *f*. In our case the feasible set F is defined by the graph of constraints, and the evaluation of this set will be explained in the next sections of this chapter.

The attributes of the tasks, in each feasible sequence being evaluated obey the following equations:

- $\mathrm{EARLIEST\_START}_i$ = max(PHASE_i + PERIOD_i * $\mathrm{INSTANCE}_i$, $\mathrm{COMPLETION}_{i-1}$), where $\mathrm{COMPLETION}_0$ is defined as zero,
- $\mathrm{COMPLETION}_i$ = $\mathrm{EARLIEST\_START}_i$ + MET_i,
- $\mathrm{DEADLINE}_i$ = PHASE_i + PERIOD_i * $\mathrm{INSTANCE}_i$ + $\mathrm{TIME\_ALLOWED}_i$,
- $\mathrm{TARDINESS}_i$ = $\mathrm{COMPLETION}_i$ - $\mathrm{DEADLINE}_i$.

2. Applicable Cost Functions

In the analysis of the set TASK of tasks two cost functions are applicable:

- Total modified Tardiness (T^s): for any sequence s of tasks we define T^s_i = T^s_{i-1} + max($\mathrm{TARDINESS}_i$,0), T^s_0 = 0, and T^s = T^s_n, where n is the total number of tasks in the sequence s [Ref Ste82 : p. 24-27];
- Maximum Tardiness (T^s_m): for any sequence s of tasks we define T^s_m = max(max($\mathrm{TARDINESS}_i$),0), for all s(i) in the sequence s.

Both cost functions above have the *recursion property*, as defined in [Ref SS86 : pp. 606-612] and [Ste82 : pp. 56-57]. The sufficient condition for the cost function is to obey the definition below:

- f(S) - min{g(f(s-j), S,j|j in R(S)}, f({}) = 0, where R(S) = {j|j in S and j has no descendats in S}, and g can be any function.

For the case of the algorithm using the enumeration techniques the recursion property is the only condition required for the cost function.

In order to use the job modular decomposition the cost function must satisfy the *job modular property*, introduced in Chapter II .[Ref. SS86 : pp. 606-612].

A more rigorous definition of the job modular property needs the introduction of a new definition: the *job module*. The definition of job module is given below:

- Let $P^1 = (J^1,R^1)$ be a poset of P=(J,R), where: J is the set of tasks in the set P, and R is the set of precedence constraints in the set P; J^1 is the set of tasks in the poset P^1, R^1 is the set of precedence constraints among the tasks of P^1; J^1 is a subet of J and $R^1 = \{(i,j)$ in R : i,j in $J^1\}$. P^1 is ε job module of P = (J,R) if and only if for every task k in $J\backslash J^1$ [7] either:

  (a) k -> i for all i in J^1, or

  (b) i -> k for all i in J^1, or

  (c) not (k -> i) and not (i -> k) for all i in J^1.

Informally the tasks in a job module are related in the same way to any job not in the module.

Using the definition above it is possible to define the job module property as follows:

- If $J^{`}$ is a job module of P = (J,R), and s^1 is an optimal sequence for the scheduling problem on $P^1 = (J^1, R^1)$, then there exists an optimal sequence s for the scheduling problem defined on P = (J,R) such that $s^1 = s|J^1$, where $s|J^1$ is the restriction of s to J^1.

Informally the job module property says that every optimal sequence for a job module is a subsequence of some optimal sequence for the entire set P.

A job module J^1 is said to be *presequenced* if an optimal sequence s^1 for the subproblem defined on (J^1,R^1) has been found, and only permutations s of J

[7] The notation $J\backslash J^1$ means $J\backslash J^1 = \{J\} - \{J^1\}$.

with $s|J^1$ are to be considered when seeking a schedule for the tasks J with precedence relation R.

The Total Modified Tardiness function does not satisfy the job module property, and the scheduling problem with the use of this cost function has been proved to be NP-complete. [Ref Ste82 : pp. 23-25]

The Maximum Tardiness cost function is an open problem in the literature searched. In order to verify if the Maximum Tardiness satisfies the job module property we will use the work developed in [Ref. MS87 : p. 22-31]. The reference cited above states that there are three sufficient conditions for the job module property to hold:

- Strong Adjacent Sequence Interchange property, a cost function f possesses the *strong adjacent interchange property* if there exists a (transitive) "preference" relation -:= defined on all pairs of sequences satisfying the following property: for all sequences s,t,u, and v, s -:= t if and only if f(u,s,t,v) <= f(u,t,s,v).

- Strong Series Network Decomposition Property, a cost function f possesses the *strong series network decomposition property* if the following conditions holds for all permutations s and t of the same set[8]: for all sequences u and v, f(s) <= f(t) if and only if f(u,s,v) <= f(u,t,v).

- Consistency Property, a cost function f with the preference relation -:= possesses the *consistency property* if the following condition holds: for all permutations s and t of the same set, if f(s) <= f(t) then s -:= t.

Let us define the notation s||v, where s is the sequence (s(1),s(2),...,s(K)), and v is the sequence (v(1),v(2),..,v(L)), then s||v is the sequence (s(1),s(2),...,s(K),v(1),v(2),...,v(L)). In order to prove the three conditions stated above for the maximum tardiness function it is necessary to show that the following condition holds:

- T^s_m is the maximum tardiness of the sequence s, T^v_m is the maximum tardiness of the sequence v, then the maximum tardiness of the sequence s||v $T^{s||v}_m$ is T^s_m or T^v_m + COMPLETION(s(k)).

[8] That means {s} = {t}.

But the analysis of the sequence s||v shows that this condition is not satisfied, as will be demonstrated below.

In order to evaluate T^s_m the following procedure must be used :

COMPLETION^s[9](s(1)) = max(0,EARLIEST_START^s(s(1))) + MET^s(s(1))

TARDINESS^s(s(1)) = COMPLETION^s(s(1)) - DEADLINE^s(s(1)),

COMPLETION^s(s(2)) = max(COMPLETION^s(s(1)),EARLIEST_START^s(s(2)))
+ MET^s(s(2)),

TARDINESS^s(s(2)) = COMPLETION^s(s(2)) - DEADLINE^s(s(2)),

.

.

.

COMPLETION^s(s(K)) = max(COMPLETION^s(s(K-1)),EARLIEST_START^s(s(K))) +
MET^s(s(K)),

TARDINESS^s(s(K)) = COMPLETION^s(s(K)) - DEADLINE^s(s(K)).

Then T^s_m = max(TARDINESS^s(s(i))), where s(i) in the sequence s.

In similar way the T^v_m is evaluated as define below:

COMPLETION^v(v(1)) = max(0,EARLIEST_START^v(v(1))) + MET^v(v(1)),

TARDINESS^v(v(1)) = COMPLETION^v(v(1)) - DEADLINE^v(v(1)),

COMPLETION^v(v(2)) =max(COMPLETION^v(v(1)),EARLIEST_START^v(v(2))) +
MET^v(v(2)),

.

.

.

COMPLETION^v(v(L)) = max(COMPLETION^v(v(L-1)),EARLIEST_START^v(v(L))) +
MET^v(v(L)),

[9] The notations COMPLETION^s(s(i)), EARLIEST_START^s(s(i)), and TARDINESS^s(s(i)), means respectively completion, earliest start, and tardiness of the task s(i) using the sequence s.

$\text{TARDINESS}^{v}(v(L)) = \text{COMPLETION}^{v}(v(L)) - \text{DEADLINE}^{v}(v(L))$.

Then $T^{v}_{m} = \max(\text{TARDINESS}^{v}(v(j)))$, where v(j) in the sequence v.

Applying the same procedure to the sequence s||v we will obtain the following:

$\text{COMPLETION}^{s||v}(s(1)) = \max(0, \text{EARLIEST\_START}^{s||v}(s(1))) + \text{MET}^{s||v}(s(1))$,

$\text{TARDINESS}^{s||v}(s(1)) = \text{COMPLETION}^{s||v}(s(1)) - \text{DEADLINE}^{s||v}(s(1))$,

$\text{COMPLETION}^{s||v}(s(2)) = \max(\text{COMPLETION}^{s||v}(s(1), \text{EARLIEST\_START}^{s||v}(s(2))) + \text{MET}^{s||v}(s(2))$,

$\text{TARDINESS}^{s||v}(s(2)) = \text{COMPLETION}^{s||v}(s(2)) - \text{DEADLINE}^{s||v}(s(2))$,

.

.

.

$\text{COMPLETION}^{s||v}(s(K)) = \max(\text{COMPLETION}^{s||v}(s(K-1)), \text{EARLIEST\_START}^{s||v}(s(K))) + \text{MET}^{s||v}(s(K))$,

$\text{TARDINESS}^{s||v}(s(K)) = \text{COMPLETION}^{s||v}(s(K)) - \text{DEADLINE}^{||v}(s(K))$,

$\text{COMPLETION}^{s||v}(v(1)) = \max(\text{COMPLETION}^{s||v}(s(K)), \text{EARLIEST\_START}^{s||v}(v(1))) + \text{MET}^{s||v}(v(1))$,

$\text{TARDINESS}^{s||v}(v(1) = \text{COMPLETION}^{s||v}(v(1)) - \text{DEADLINE}^{s||v}(v(1))$,

$\text{COMPLETION}^{s||v}(v(2)) = \max(\text{COMPLETION}^{s||v}(v(1), \text{EARLIEST\_START}^{s||v}(v(2))) + \text{MET}^{s||v}(v(2))$,

.

.

.

$\text{COMPLETION}^{s||v}(v(L)) = \max(\text{COMPLETION}^{s||v}(v(L-1)), \text{EARLIEST\_START}^{s||v}(v(L))) + \text{MET}^{s||v}(v(L))$,

$\text{TARDINESS}^{s||v}(v(L)) = \text{COMPLETION}^{s||v}(v(L)) - \text{DEADLINE}^{s||v}(v(L))$.

Then $T^{s||v}_{m}$ = max(TARDINESS$^{s||v}$(s(i)), TARDINESS$^{s||v}$(v(j))), where s(i) and v(j) in the sequence s||v.

The analysis of the definitions of the parameters utilized in the equations above shows that deadline, maximum execution time, and earliest start of the tasks are completely independent of the sequence or the concatenation of sequences, that is the deadline, maximum execution time, and earliest start of all the task in s and in v remains the same in the sequence s||v.

The completion of the tasks belonging to the sequence s remain the same in the sequence s||v, but the completion of the tasks in the sequence v do not remain the same in the sequence s||v. The completion time of a task v(j) in the sequence s||v is always equal or greater than the completion time of this same task in the sequence v. The difference between the these two completion times is not a constant for all tasks in v. As an example lets assume that the task v(j-1), in the sequence s||v, had its completion time increased by an amount of time X, then the task v(j), in the sequence s||v, has the following possibility of interest for our analysis[10]:

COMPLETION^{v}(v(j-1)) < EARLIEST_START^{v}(v(j)), and

COMPLETION^{v}(v(j-1)) + X > EARLIEST_START^{v}(v(j)), then

COMPLETION$^{s||v}$(v(j)) = COMPLETION^{v}(v(j)) + X - COMPLETION^{v}(v(j-1)).

The possibility shown does not satisfy the condition that the tardiness of the tasks in v are modified by a constant and invalidates our basic premise to try the proof that the maximum tardiness function holds the three sufficient conditions.

As defined in [Ref. MS87 : pp. 22-31] these conditions are sufficient, but not necessary then we can not prove that the maximum tardiness has not the job

[10] There are more possibilities, but four of them are unfeasible, and the other four possibilities satisfy COMPLETION$^{s||v}$(v(j)) = COMPLETION^{v}(v(j)) + constant.

module property. In order to prove that the maximum tardiness does not have the job module property we need to reduce the whole problem to any know N-complete problem (as was done in the case of the Total Weighted Tardiness in the [Ref. LR78 : pp. 23-34]).

The maximum tardiness cost function satisfies the adjacent pairwise job interchange property defined in [Ref. Smi56] for some cases analized. The pairwise interchange property says that if two tasks are not in the preference order they may be changed without affecting the cost function, it is only affected when we interchange tasks that are in the preference order.

The reduction proposed above is out of the scope of this thesis, and will be not done. The question of whether the maximum tardiness cost function has the job module property is left open.

3. Selected Cost Function

The analysis of the two cost functions available for our scheduling problems reveals to us the following facts:

- both cost functions are applicable to the evaluation of the enumeration techniques,
- the total modified tardiness is not applicable to the job modular decomposition,
- the maximum tardiness is an open question with concerns its application to the job modular decomposition,
- the maximum tardiness cost function locates the task with the maximum tardiness,
- the total modified tardiness cost function furnish the total tardiness of the system.

In our opinion the best option is to select the maximum tardiness, because we have immediate access to the task with maximum tardiness, and also because we can not discard the possibility that this cost function may apply to the job modular decomposition.

Our approach, based on the data available, will be to develop with details the enumeration techniques algorithm to find an optimal algorithm for the scheduling problem, and just to develop the basic guidelines for the implementation of the algorithm for the scheduling problem using the job modular decomposition.

D. THE ENUMERATION TECHNIQUES OPTIMAL SCHEDULING ALGORITHM

Two new concepts are necessary now, we define a sequence as a *legal* if it satifies the precedence constraints represented by the graph of constraints, and as a *feasible* if and only if it satisfies simultaneously the precedence constraints represented by the graph of constraints and the timing constraints.

The development of the optimal scheduling algorithm is based on the enumeration of all the legal sequences of the tasks in the graph of constraints. The enumeration of all the legal sequences may be explicit or implicit.

In the first case all the legal sequences are defined before we verify if any of them is feasible. In this approach the number of legal sequences can be very large and we may not be able to afford to store them all. The advantage is that we may observe the behavior of the cost function for each legal sequence, as well as the relations among the tasks. This fact can be useful in future work about how to obtain a feasible sequence for an unfeasible system, doing the minimum of modifications on the system.

The implict approach generates one legal sequence at time and verifies if it is feasible, when a feasible sequence is find or all the legal sequences had been generated the proccess is halted. The only sequence that must be stored is the current sequence. The advantages of this approach are the economy of storage space and the possibility to apply a dynamic programming method to the evaluation of the cost function. This method does not allow a deep analysis of possible modifications if the system does not have a feasible solution.

In this work we introduce both approaches.

1. Explicit Enumeration

The explicit enumeration is obtained using the maximum legal sequence in the lexicographic order. In order to obtain the maximum lexicographic order legal sequence we mapped the graph of constraints into the natural numbers in the section C of this chapter.

In this thesis we use the topological sort described in [Ref Mar88], modified in order to obtain the maximum lexicographic order legal sequence.

The steps necessary to obtain the the optimal enumeration scheduling algorithn, in this approach, are:

- Obtain the ancestors and descendants of each task,
- Obtain the maximum lexicographic order legal sequence,
- Generate all the possible legal sequences,
- Apply the cost function to each legal sequence generated, until one of them is feasible (optimal).

We define, in this work, the *set of successors* of a task i all the tasks that obey the relation i -> j. The definition of *set of predecessors* of a task i is the set of all the tasks j such that j -> i.

Another definition necessary is the concept of ancestor and descendant of a task. We define as ancestor of a task *k* all the tasks *i, j, ...* such that exists a precedence relation that obeys the following condition: i -> j -> ... -> k. More than one chain may contains ancestors of a task k. The definition of descendants is the reverse of the definition of ancestors, that is: in the scheme described before *j, ..., k* are descendants of the task *i;* the same observation about the chains applies to the case of the descendants.

a. *Ancestors and Descendants*

The generation of the ancestors and descendants of each task is constructed upon the predecessors and successors of each task.

By the construction of the graph of constraints the following sets of ancestors and descendants are already defined:

- the set of the ancestors of TASK(1) is the empty set, and the set of ancestors of the TASK(TASK_LENGTH) is the set TASK\TASK(TASK_LENGTH),
- the set of the descendants of TASK(1) is the set TASK\TASK(1), and the set of descendants of the TASK(TASK_LENGTH) is the empty set.

We selected as structure to hold the set of descendants and antecessors of each task two sets (ANCESTORS(I) and DESCENDANTS(I)).

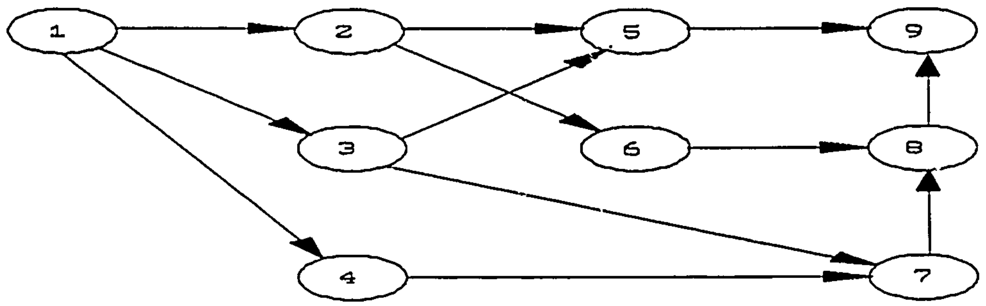
We describe first the construction of the set ANCESTORS. Assume that we are evaluating the antecessors of the task i then we begin by including in the set ANTECESSORS(i) all the predecessors of the task i, after this we include in this list all the ancestors of all the predecessors of the task i. The construction of the DESCENDANTS of the task i is done in a similar fashion, we start including in the set DESCENDANTS(i) all the successors of the node i, after it is done we include in the set all the descendants of all the successors of the task i.

The description of these algorithms is available below. Figure 25, on page 76, illustrates an example of the application of the algorithm to find sucessors and predecessors. Figure 26, on page 78, illustrates the application of the algorithm to find ancestors and descendants.

a1. *Algorithm for Sucessors and Predecessors*

```
find_sucessors_predecessors;
    for I in 1 .. TASK_LENGTH loop
        SUCESSORS(I) := {};
        PREDECESSORS(I) := {};
```

```
            for J in 1 .. TASK_LENGTH loop
                if E_TASK(I,J) = 1 then include J in SUCESSORS(I)
                end if;
        end loop;
        for J in 1 .. TASK_LENGTH loop
                if E_TASK(J,I) = 1 then include J in PREDECESSORS(I);
                end if;
        end loop;
    end loop;
end find_sucessors_predecessors.
```



GRAPH OF CONSTRAINTS

TASK	SUCESSORS	PREDECESSORS
1	{2,3,4}	{ }
2	{5,6}	{1}
3	{5,7}	{1}
4	{7}	{1}
5	{9}	{2,3}
6	{8}	{2}
7	{8}	{3,4}
8	{9}	{6,7}
9	{ }	{5,8}

Figure 25 Sucessors and Predecessors

a2. Algorithm for Ancestors and Descendants

```
find_ancestors_descendants;
   for I in 1 .. TASK_LENGTH loop
       ANCESTORS(I) := PREDECESSORS(I);
       DESCENDANTS(I) := SUCESSORS(I);
   end loop;
   for I in 1 .. TASK_LENGTH loop
       for P in PREDECESSORS(I) loop
          ANCESTORS(I) := ANCESTORS(I) U ANCESTORS(P);
       end loop;
   end loop;
   for I in reverse 1 .. TASK_LENGTH loop
       for S in SUCESSORS(I) loop
          DESCENDANTS(I) := DESCENDANTS(I) U DESCENDANTS(S);
       end loop;
   end loop;
end find_ancestors_descendants.
```

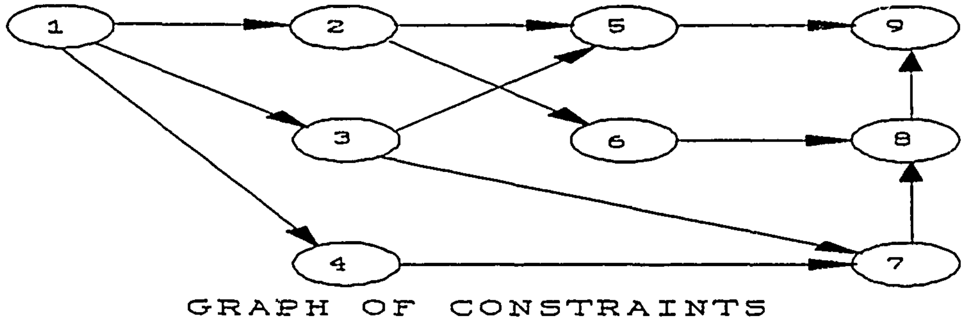
b. Maximum Lexicographic Order Legal Sequence

As said before the construction of the maximum lexicographic sequence is done using a modified topological sort algorithm, the description of the algorithm is furnished below, and the result obtained using the graph of constraints shown in Figure 25, on page 76, is the following: (1,4,7,3,2,6,8,5,9).

b1. Algorithm for Maximum Lexicograhic Order Legal Sequence

```
remove;
  g.nodes := g.nodes\n;
  g.edges := {(i,j) in g.edges| i<>n and j<>n};
```

```
end remove;

maximum_lexicographic_sequence;

  g.nodes := {1, ..., TASK_LENGTH};

  g.edges := E_TASK;

  while not g.nodes = {} loop

     n := max{v in g.nodes|PREDECESSORS(v) ={}};

     MAX_LEX_SEQ := MAX_LEX_SEQ || {v};

     g := remove(g,n);

  end loop;

end maximum_lexicographic_sequence.
```



TASK	ANTECESSORS	DESCENDANTS
1	{}	{2,3,4,5,6,7,8,9}
2	{1}	{5,6,8,9}
3	{1}	{5,7,8,9}
4	{1}	{7,8,9}
5	{1,2,3}	{9}
6	{1,2}	{8,9}
7	{1,3,4}	{8,9}
8	{1,2,3,4,6,7}	{9}
9	{1,2,3,4,5,6,7,8}	{}

Figure 26 Ancestors and Descendants

2. Implicit Enumeration

The algorithm for the implicit enumeration technique is presented in the section D.3.b, together with the algorithm for the explicit enumeration technique.

This approach allows further development in order to include the evaluation of the cost function using dynamic programming technique. This work is not developed in this thesis. It will be necessary to evaluate the phases of the operators during the generation of the legal sequences. Another possibility is to include a branch and bound methodology in this approach.

The current version of this approach requires the following steps:

- obtain the predecessors of all the tasks in the graph of constraints,
- obtain a legal sequence for the graph of constraints,
- evaluate the legal sequence obtained.

3. Legal Sequences

The two methods to obtain all the legal sequences are presented above. The implicit enumeration technique requires the use of the concept of predecessors. All the legal sequences are obtained from the ordered graph of constraints. The explicit enumeration technique utilizes the concepts of ancestors and descendants and it checks the maximum lexicographic order legal sequence and tries to find all the legal sequences smaller than it until it to reaches the sequence (1, 2, ..., TASK_LENGTH). Both algorithms are described below.

a. *Algorithm for Implicit Enumeration*

```
recursive_legal_sequences;
    if g is empty then generate []
    else for each node n in g such that predecessor(n) = {} loop
      for each sequence s in feasible_sequence(g - {n})
         generate {n} || s;
    end loop;
  end loop;
end recursive_legal_sequences.
```

b. Algorithm for Explicit Enumeration

```
sequential_legal_sequences;
  current sequence := maximum lexicographic order legal sequence;
  include sequence in legal_sequences;
  while sequence > (1,2, ..., TASK_LENGTH) loop
     generate the next lexicograph small legal sequence than
     current sequence;
     current sequence := new sequence;
     include current sequence in legal_sequences;
  end loop;
end sequential_legal_sequences.
```

4. Evaluation of the Cost Function

The evaluation of the cost function for each feasible permutation is a straight-forward application of the equations defined in section D. The only precaution necessary is to verify if the task being analyzed is the first instance of the corresponding operator. If it is the first instance then it is necessary to define the phase of the corresponding operator to be the start time of the task under analysis.

All the legal sequences are enumerated and the cost of each legal sequence is evaluated. The lowest cost found so far and the sequence that realizes the lowest cost are maintained in program variables.

The process can stop as soon as a sequence with a cost less than or equal to zero is found, since such sequence represents a feasible sequence, or in other words a feasible static schedule for the scheduling problem.

The details of the algorithm are described below.

a. Algorithm for Evaluate the Maximum Total Tardiness

```
evaluate_sequences;
  best_cost := maximum_integer;
  for each legal sequence s loop
      c:= cost(s);
      if c < best_cost then
          best_cost := cost;
          best_sequence := s;
      end if;
      if c =< 0 then exit;
  end loop;
  if best_cost =< 0 then
          build_schedule(best_sequence);
  else
          error_message(best_cost,best_sequence);
  end if;
end evaluate_sequences.
```

5. Summary of the Optimal Enumeration Scheduling Algorithm

This algorithm guarantees that if at least one solution of the problem under analysis exists then it will be discovered.

Two possible DFD are available for the enumeration scheduling algorithm, depending on wich enumeration technique is selected.

The DFD of the algorithm are illustrated in Figure 27 and Figure 28, on page 82.

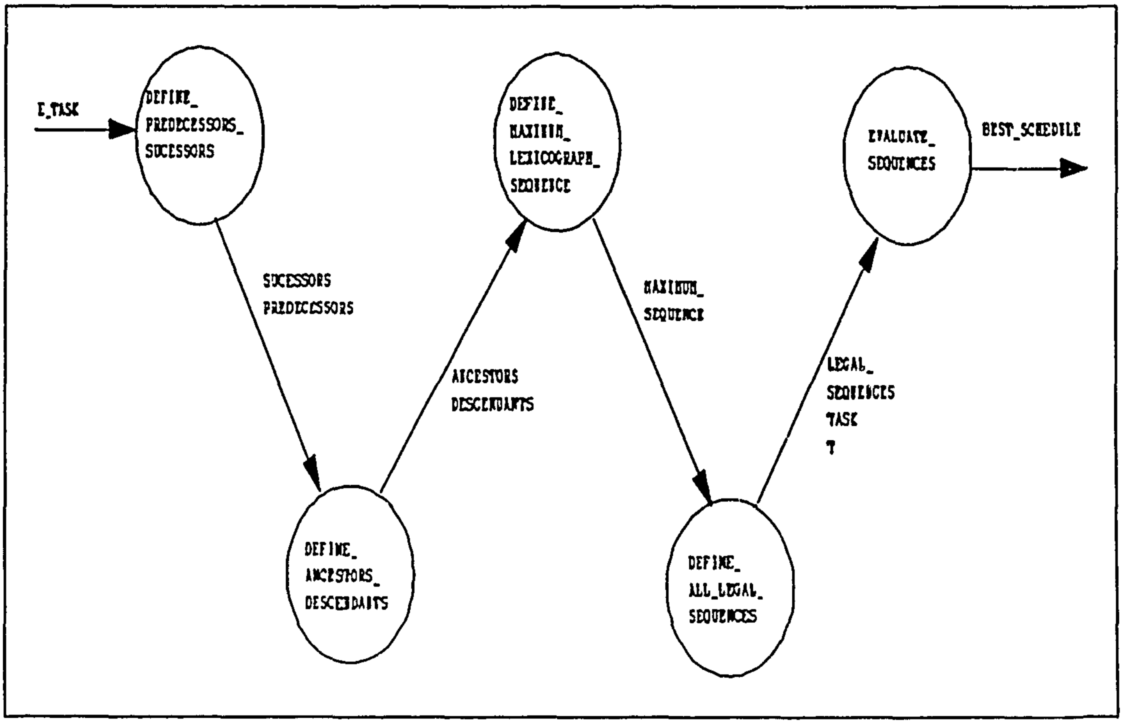


Figure 27 1st Level DFD Explicit Enumeration

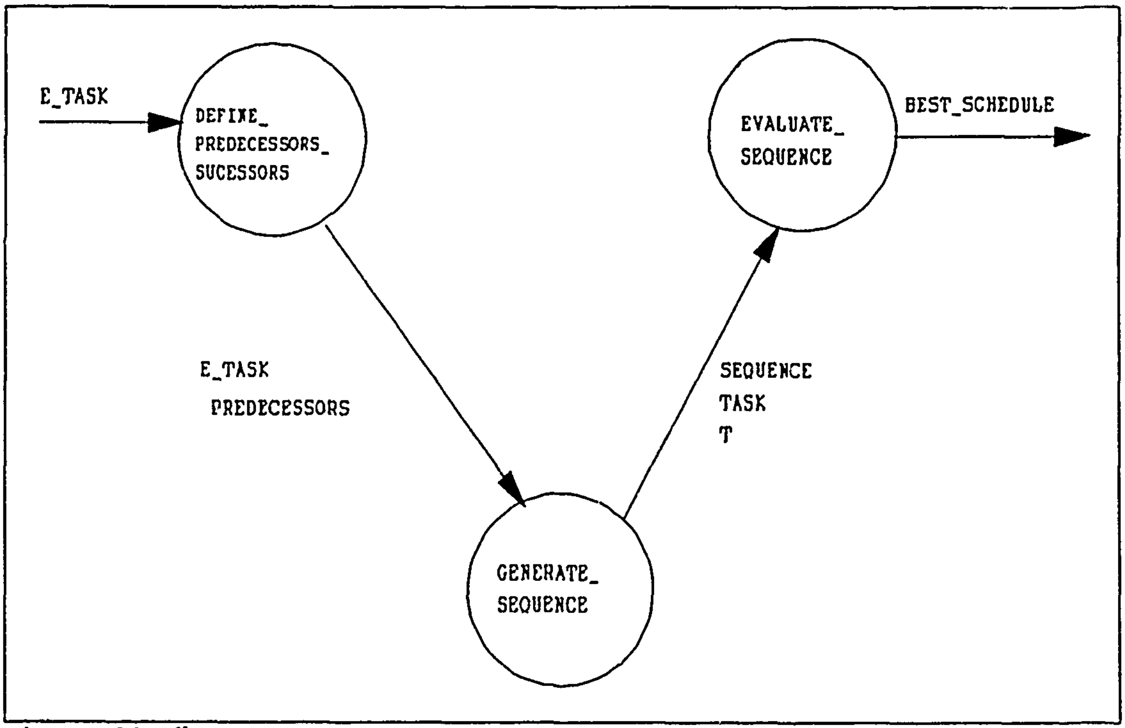


Figure 28 1st Level DFD Implicit Enumeration

E. THE JOB MODULAR DECOMPOSITION ALGORITHM

As cited in the section C, we can not ensure the optimality of this algorithm, but it has the advantage that converges to a feasible permutation more quickly than the enumeration algorithm, if it succeeds.

Another important aspect of this algorithm is that it uses the most recent developments in the area of analysis of networks[11], and some of the ideas may be of interest for the case of multiple processors.

The approach selected to introduce this algorithm is to describe the procedures necessary, without discussing its respective data structures or step by step development; all the surveyed possible alternatives for each procedure are cited and the reason why we selected a specific one is explained.

The basic steps to generate a scheduling algorithm by the job modular decomposition are the following:

- generate of the graph of constraints as described in section III.B,
- obtain the tree decomposition of the graph of constraints in the corresponding job modules,
- evaluate the possible phases of the operators,
- presequence the job modules recursively in order to obtain the global solution for the scheduling problem, for each possible set of phases.

1. Tree Decomposition of the Graph of Constraints into Job Modules

The decomposition of the graph of constraints in a tree of job modules requires some concepts not yet discussed in this thesis.

We will introduce the new concepts necessarys, as well as a deeper insight into the basis of the job modular decomposition, and after this we will point out where the available algorithms may be found.

[11] The other important area in the analysis of network is the concept of Petri Nets introduced in Chapter II.

a. *Additional Concepts*

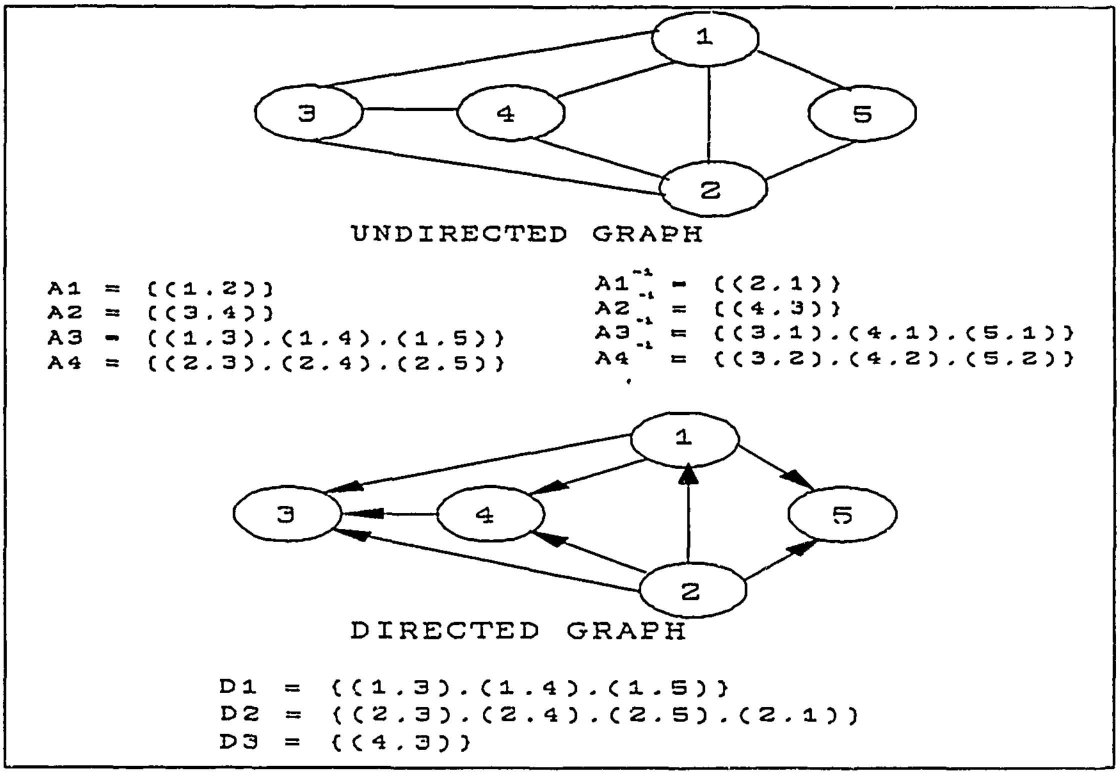
First we must define the concept of the transitive orientation of a graph: in order to possesses a transitive orientation an undirected graph must have a finite number of vertices and edges, with no edge joining a vertex to itself and no distinct edges joining the same pair of vertices. The undirected graph will be defined by G = (V,E), where V is the set of vertices and E is a set of unordered pair (i,j), which represents the edges joining the vertex i to vertex j (or vice-versa). An orientation of G is an assignment of an unique direction *i* -> *j* or *j* -> *i* to every edge (i,j) in E. The resulting directed image of G is denoted G^d where $G^d = (V,E^d)$, and E^d is now a set of ordered pairs (i,j), where (i,j) implies that i -> j. Transitive orientable graphs have the property that they admit a labeling v_1, v_2, ...,v_n of their vertices under which for $i < j < k$ the existence of an edge joining v_i to v_j and one joining v_j to v_k implies the existence of an edge joining v_i to v_k. [Ref. PLE71 : pp. 160-175] For more details about transitive orientation of a graph, other papers of interest [illegible] [SF70 : pp. 648-667], [Ref. PE72 ; pp. 400-410], and [Ref. Spi85 : pp. 658-670] A transitive orientable graph G, that has the oriented image G^d, is called a comparability graph.

The second concept necessary is the implication classes of the [illegible] of an undirected graph. Let G = (V,E) be a comparability graph and |^ be a bin[illegible] relation defined as follows:

- (i,j) |^ (i',j') , if and only if i = i' and (j, j') not in E^d, or j = j' and (i,i') not in E.

Figure 29, on page 85, illustrates the concept of implication class.[Ref. Gol77b]

The reflexive, transitive closure |^* of |^ is an equivalence relation on E and hence partitions E into what is defined as the implication classes of E. Thus edges (i,j) and (m,n) are in the same implication class if and only if there exists a |^-chain of edges $(i,j) = (i_0,j_0)$ |^ (i_1,j_1) |^ ... |^ $(i_k,j_k) = (m,n)$, with k >= 0. A complete definition and explanation about implication classes as well about comparability graphs can be found in [Ref. Gol77a : pp. 68-90], [Gol77b : pp. 199-208], [Ref. SF70 : pp. 648-667], [Spi85 : pp. 658-670], and [Ref. Ste82 : pp. 166]. An algorithm to find the implication class of an undirected graph is available in [Ref. Gol77b]; the same algorithm, modified to work on directed graph, is illustrated in [Ref. Ste82 : p. 166].



A1 = {(1,2)}
A2 = {(3,4)}
A3 = {(1,3),(1,4),(1,5)}
A4 = {(2,3),(2,4),(2,5)}

$A1^{-1}$ = {(2,1)}
$A2^{-1}$ = {(4,3)}
$A3^{-1}$ = {(3,1),(4,1),(5,1)}
$A4^{-1}$ = {(3,2),(4,2),(5,2)}

D1 = {(1,3),(1,4),(1,5)}
D2 = {(2,3),(2,4),(2,5),(2,1)}
D3 = {(4,3)}

Figure 29 Implication Class

b. Job Modular Decomposition Basis

Let $P_0 = (J_0,R_0)$ be a poset on m elements with $J_0 = \{i_1, i_2, ..., i_m\}$ and let $P_h = (J_h,R_h)$ for h = 1, 2, ..., m be disjoint posets. The *composition poset* P = (J,R) is defined by $J = U^m_{h=1} J_h$ and $R = U^m_{h=1} R_h$ U {(i,j): i in J_h, j in J_k and (i_h,i_k) in R_0}. For this composition we use the notation P = [P_1, ..., P_m] and refer to P_0 as the *outer factor* and P_1, ..., P_m as the *inner factors*. We say that P is the *series decomposition* when P_0 is a chain and the *parallel decomposition* when P_0 is an antichain. In any other case, P is called an *N-composition* (for "neighborhood" composition).

Each inner factor is called a *module* of P. P is said to be *decomposable* if it contains a non-trivial module; otherwise P is *indecomposable* or *prime*.

The definitions above are the basis of all the job modular decomposition. The graph cf constraints developed in this work has the property that P_0 is always a series module in relation to the inner factors. The inner factors defined in this subsection are the basic sequences that need to be sequenced in order to obtain the final sequence for the global scheduling problem. Figure 30, on page 87, illustrates the concepts of outer and inner factors. In Figure 30, the root of the tree decomposition is the outer factor, and all the other nodes are the inner factors.

c. Algorithms Available for the Job Modular Decomposition

In our survey we found five basic approaches to the job modular decomposition, which are described in the following papers: [Ref. MJ89 : pp. 1-19], [Ref. Cun82 : pp. 214-228], [Ref. Sid81 : pp. 190-204], [Ref. MB83 : pp. 170-184], and [Ref. Ste82 : pp. 158-167]. The first reference presents the fastest algorithm (O(n^2)) but is the most difficult in terms of implementation. All the other four algorithms have of the same time complexity (O(n^3)). We selected the last one because it is the simplest to implement, and also because it is the one with the most detailed description of the data structures necessary as well all the details of the procedure. The global description of the algorithm, as it is described in [Ref. Ste82: pp. 181 - 183], is reproduced:

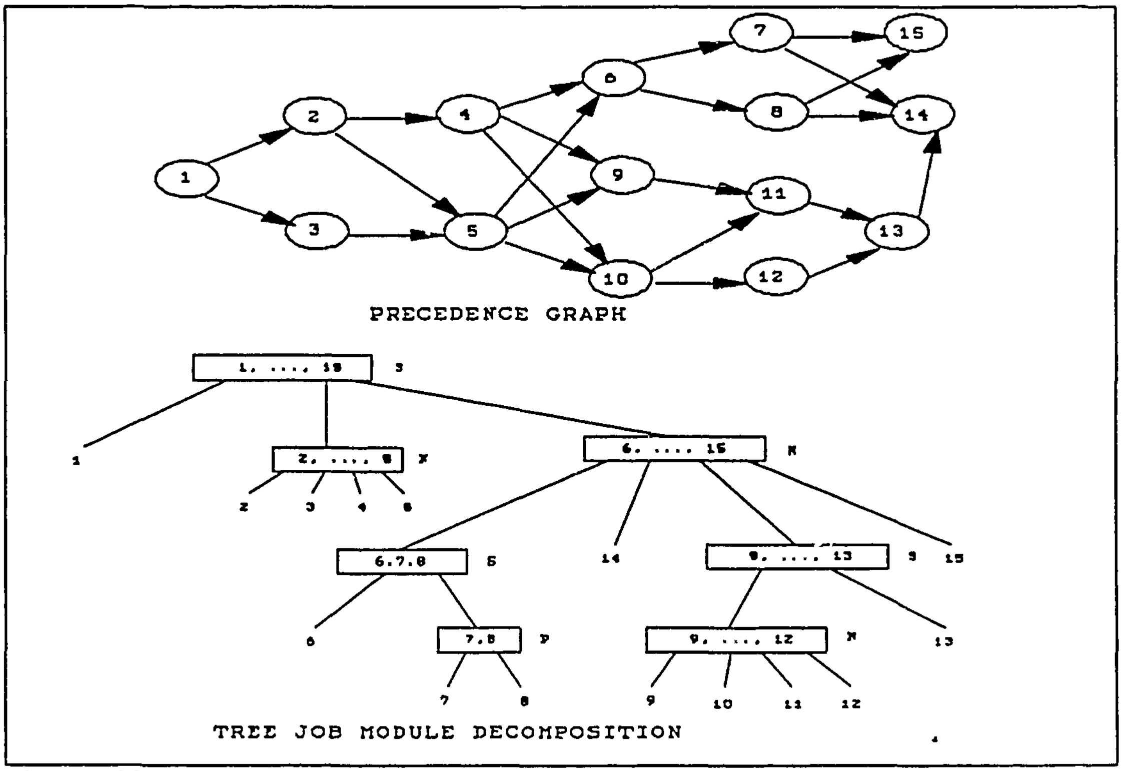


Figure 30 Job Modular Decomposition

"Consider the acyclic transitive digraph G = (V,A). In the k^{th} iteration of the algorithm we denote the current graph by $H_k = (V_k, A_k)$, and enumerate an implication class of H_k, denoted by B_k. B_k will define a decomposition $H_k = H_{k+1} [I_{k,1}]$ of H_k. After this we find a prime decomposition of $I_{k,1}$ and then carry on with the (k+1)-st iteration to decompose H_{k+1}.

0. Let k = 1, B_1 = { }, H_1 = V, A_1 = A, FLAG = 0.

1. If FLAG = 0 select an (i,j) in A_k, such that i is an immediate predecessor of j in H_k. If FLAG = 1 select an (i,j) in $A_k \setminus B_{k-1}$, such that i is an immediate predecessor of j in H_k and let FLAG = 0. If no such (i,j) exists, check whether any intermediate g.:aphs are still in STORAGE, if the answer is no, STOP, if the answer is yes let H_k be the last intermediate graph in STORAGE, delete this graph from STORAGE and go to the beginning of 1.

2. Find the implication class B_k in H_k, defined by the arc (i,j), and let the set of vertices spanned by B_k in H_k be Y_k.

3. Y_k is a job-module in H_k. Output Y_k to the job-module list, along with the composite vertex v^*_k, which will replace it in H_{k+1}. If $|Y_k| = 2$ go to 6, otherwise let $I_{k,1} = (Y_k, B_k)$. If $I_{k,1}$ contains every arc of the induced subgraph (Y_k, B_k) go to 4, otherwise let FLAG = 1 and go to 6.

4. Let r = 1. Enumerate the pairs of vertices in Y_k, until one pairs, say u,v in Y_k is found, such that $\{u,v\}$ is a job-module in $I_{k,1}$. Output $\{u,v\}$ to the job-module list, along with the composite vertex w^*_{kr} representing it. If no such u,v exist $I_{k,1}$ is indecomposable and go to 6, otherwise go to 5.

5. $\{u,v\}$ defines a decomposition of the subgraph $I_{k,r}$. Let this decomposition be $I_{k,r} = I_{k,r+1}\ [J_{k,r+1}]$, where $I_{k,r+1} = (Y_{k,r+1}, B_{k,r+1})$, $J_{k,r+1}$ is the subgraph of $I_{k,r}$ induced by $S_{k,r+1} = \{u,v\}$, $Y_{k,r+1} = Y_{k,r} \setminus S_{k,r+1}$ U $\{w_{kr}\}$ and $B_{k,r+1} = \{(a,b)|(a,b)$ in $B_{k,r}$ and a,b not in $\{u,v\}$ U $\{(a,w_{kr})|a$ in $Y_{k,r+1}$ and (a,u) in $B_{k,r}\}$ U $\{(w^*_{kr},b)|b$ in $Y_{k,r+1}$ and (u,b) in $B_{k,r}\}$.

Continue the enumeration of the pairs of elements in $Y_{k,r+1}$ (considering only those which have not been looked at yet) until one pair, denoted $\{u,v\}$, is found, such that $\{u,v\}$ is a job-module in $I_{k,r+1}$. If no such pair can be found, the graph $I_{k,r+1}$ is indecomposable, go to 6. Otherwise let r = r + 1, output $\{u,v\}$ to the job-module list, along with the composite vertex $w^*_{k,r}$ representing it and go to the beginning of step 5.

6. Let $H_{k+1} = (V_{k+1}, A_{k+1})$ be defined as follows:

$V_{k+1} = V_k \setminus Y_k$ U $\{v^*_k\}$. $A_{k+1} = \{(a,b)|(a,b)$ in A_k, a not in Y_k, b not in $Y_k\}$ U $\{(a,v^*_k)\backslash a$ in $V_k \setminus Y_k$ and there is b in Y_k s.t. (a,b) in $B_k\}$ U $\{(v^*_k,a)|a$ in $V_k \setminus Y_k$ and there is b in Y_k s.t. (b,a) in $B_k\}$. If FLAG = 1 go to 8, otherwise replace H_k by H_{k+1} and go to 7.

7. Let k = k + 1, $B_k = \{\}$ and go to 1.

8. The subgraph contains an implication class of H_k other than B_k. Store the graph H_{k+1} in an area called STORAGE, let $H_{k+1} = I_{k,1}$ and go to 7."

The composite vertex cited in the algorithm above is just a indicator of the outer factor (job-module), and replaces the outer factor in the new graph. [Ref. Ste82 : pp. 175-176]. At the end of the process we will have only one composite vertex in the resultant graph, which is the tree job modular decomposition of the original graph.

2. Evaluation of the Possible Phases of the Operators

The sequencing of the tasks inside each inner factor (or job module) is done using the cost function. This computation needs to have all the data necessary (parameters for the equations defined in III.C.1) at hand, but the phase of an operator is an important parameter that is only available after the first instance is scheduled, then we must to find a way to solve this problem.

The best solution that we found is to apply the algorithm described in III.D.2.b (originally used to find maximum lexicograph order legal sequence of the graph of constraints) to the precedence graph, and obtain what we defined as the *phase generator*. With the phase generator defined we then apply the algorithm described in III.D.3.b (originally defined to find all the maximum legal sequences of the graph of constraints). This procedure furnishes all the legal orderings for the first firing of each operator, one for each feasible permutation of the phase generator under the restrictions of the precedence constraints. Since the first firing of each operator defines the phase of the operator, this solves our problem.

Thus we generate all possible phases, and pass them as a parameter to a subprogram which does optimal sequencing. This is done repeatedly until we find a feasible solution, keeping track of the lowest cost and best sequence as in the enumeration algorithm of section III.D.

3. Presequencing the Job Modules

For each set of possible phases of the operators we must presequence the job modules obtained in III.E.1.c.

The presequencing of the job modules must start at the leaves of the tree decomposition. The first step is to presequence each leaf module. Intially all of the leaves contain sequences of length one, which are already pre-sequenced.

The next step is to combine two presequenced leaves in order to obtain a new leaf that is the union of them. The procedure that we recommend is to find which leaf has the smallest TASK.ID, generate a new leaf that is the union of the sequence of the leaf that has the smallest task followed by the sequence of the other leaf. Then we apply the algorithm described in III.D.3, but only on the elements that belong to the former leaf at the right of the new leaf, after all feasible permutations are obtained we apply the cost function and save the best case. This procedure must be done recursively until we have all the leaves eliminated and reach the root. The current sequence is the final sequence for the scheduling problem, and the current maximum tardiness is the final result of the scheduling problem.

F. SUMMARY

In this chapter we introduced two important algorithms :

- generation of the graph of constraints,
- the optimal scheduling by enumeration techniques.

The first is a tool that allows the development of the optimal scheduling algorithm using the enumeration techniques, and is the first algorithm that tries to combine in just one piece information the precedence constraints and the timing constraints. The merging of information is just partial, because only the period of the operators is taken in account and the synchronization is restricted.

The optimal static scheduling algorithm using enumeration techniques is a more effective approach to the scheduling problem with precedence constraints and represents an advance in the current version of the algorithms being utilized by the CAPS system. It can not be considered a quick algorithm, but is a reliable algorithm in the sense that its construction is simple with respect to structures and concepts utilized. Another important aspect of this algorithm is the optimality guaranteed by the exhaustive analysis of all the possible feasible permutations of the graph of constraints.

In our point of view the weakness of this algorithm is the restriction imposed on the sporadic operators (must be converted to the equivalent periodic operators). This reduces the set of feasible solutions of the actual problem. Some possible ways to enlarge the set of feasible solutions to the actual problem are addressed in the next chapter.

The other important aspect discussed in this chapter is the definition of the basic guidelines for the utilization of the theory of job modular decomposition in the analysis of the scheduling problem with precedence constraints.

During this work it was neither possible to prove the optimality of this approach, nor its non optimality, but the data collected seems to enforce the position that it may be used at least as one heuristic algorithm.

IV. ANALYSIS AND COMMENTS

The basic objective of this chapter is to evaluate some points about the algorithms developed in the previous chapter and to discuss some modifications to the CAPS system that may improve its range as well introduce new concepts which may be utilized during future work on the development of new algorithms for the scheduling problem in a single processor or in the case of multiprocessors.

A. EVALUATION OF THE ALGORITHMS

During the design of the optimal enumeration scheduling algorithm we showed that its is strongly based on two basic concepts:

- the graph of constraints,
- evaluation of the basic sequence.

The optimality of the algorithm is guaranteed by the theory about feasible subsets defined in [Ref. Ste82], only under the constraints defined by the graph of constraints. The graph of constraints represents one abstraction of the actual problem, two major simplifications are introduced in it: the assumption that all the operators are periodic, and a simplified scheme for synchronization among the tasks. The analysis of how to handle the sporadic operators is on Section IV.B.

During the definition of the criteria to be used to synchronize the operators we made a tradeoff between how much synchronization to define among the operators and the size of the set of feasible solutions to the scheduling problem. Our analysis of the problem showed that when the level of synchronization is increased there is a reduction in the number of possible solutions. This fact is a consequence of the increased number of constraints that must be satisfied. As a middle point we adopted the criteria that the user must define operators with the same (or multiples / submultiples) periods when she/he wants perfect synchronization.

Another important decision made, for the construction of the graph of constraints, was not to allow the existence of external inputs. This restriction does not represent a loss of generality, it just implies that any external input to the system must be represented as an operator in the graph of constraints. This means that we are forcing the prototype designer to include in the system a simulation of the producer operator that triggers the sporadic operators. The producer operator may be simulated using the *delay* statement defined in Ada, and a predefined random variable (assuming that the user knows the behavior of the triggering operator: mean, deviation, and type of distribution). To avoid burdening the user with extra work the reusable software database of CAPS may include some generic modules for the producer operator. These patterns need to leave to the user the definition of the parameters which will characterize the desirable function distribution that best fit on the operator.

During this work we did not evaluate the time complexity or space complexity of the algorithm. However it is not a linear function of the size of the input, assuming as size of the input the number of operators to be scheduled. This fact is evident for the construction of the graph of constraints, as we can see in Figure 31, on page 94. The set of operators is the same as described in the Figure 20, and the precedence constraints are also the same, but the periods are different. This simple modification is enough to turns a simple scheduling problem into another problem with a greater number of constraints that must be obeyed (in the former case we had 14 tasks to be scheduled and now we have 36 tasks that must be schedule, the harmonic block length jumped from 30 to 56), as illustrated in the figure.

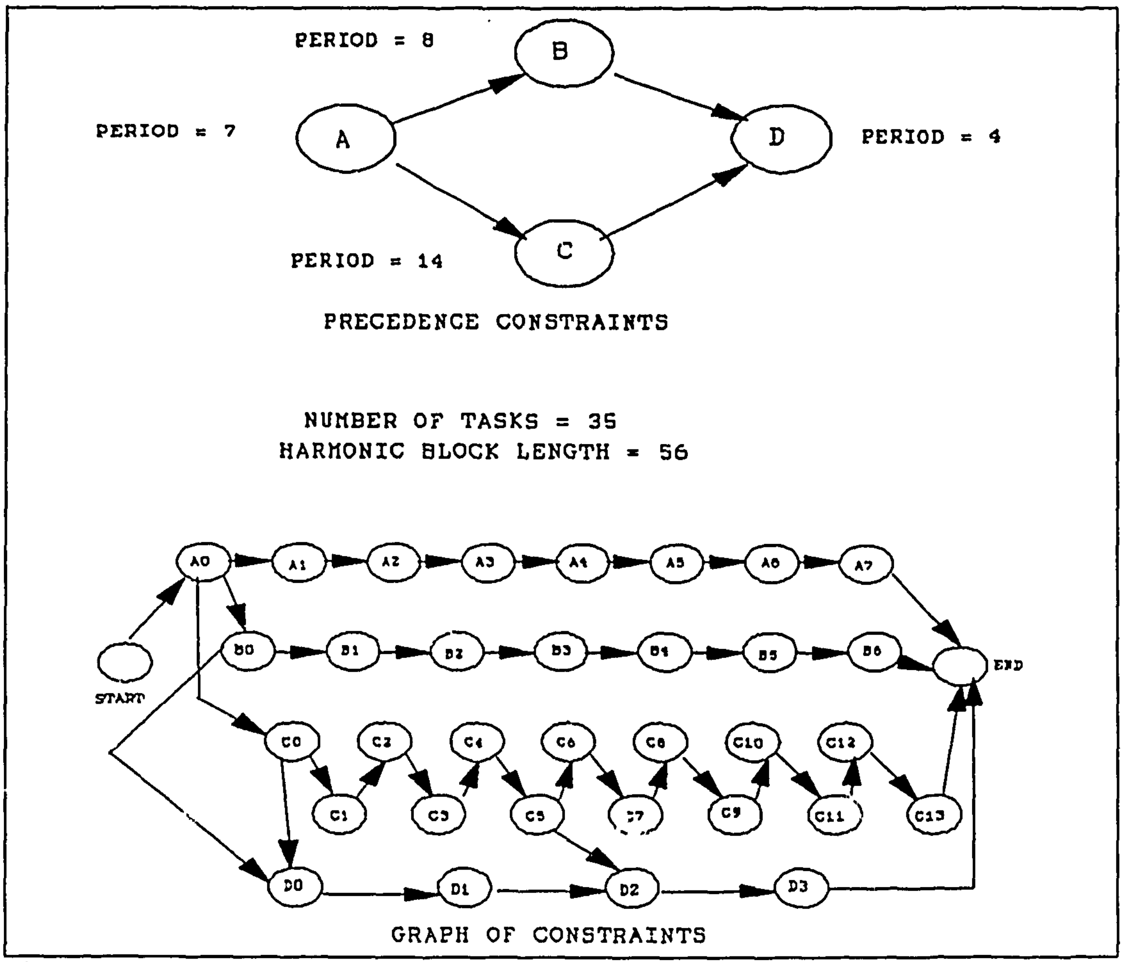


Figure 31 Influence of the Period on the Graph of Constraints and the LCM

In this work we will not define the explicit function of the time and space complexity of the algorithm. The best evaluation that may be done now is that the overall time complexity is defined by three major steps in the algorithm: the interconnection of the chains of tasks, the evaluation of the maximum lexicographic order legal sequence , and the evaluation of all the legal sequences. The interconnections of the chains of tasks is no greater than O(n^2); the evaluation of the basic sequence is O(kn), where k is the maximum number of task without predecessors during the topological sort; the evaluation of all the feasible permutations of the basic sequence is O(p!) , where p is the maximum difference from the sequential position to the shift to the left of all the tasks. It is important to highlight that this worst case is not likely to occur because it implies that all the tasks are simultaneously occupying the maximum leftmost position. The major factors that determine the complexity of the algorithm are: number of operators, and periods of the operators. The precedence constraints influence the time complexity, since they determine the size of the set of feasible solutions. It is difficult to characterize the distribution of precedence constraints expected for practical applications.

We may have more than one feasible sequence of the set of tasks that obey the timing constraints, but all of these solutions are equally desirable. This is based on the fact that we have a well defined interval that will be repeated (the harmonic block length) and the time that the critical tasks will use is always the same in all the solutions, so that the free time available to be allocated to the non-critical operators is the same in all the solutions.

We assumed in the development of this thesis that all the precedence constraints furnished by the CAPS are well formed and we do not include any kind of check on it. The reasons behind this assumption is the same employed in the "warning" message of any compiler, because there are many possible precedence constraints constructions that are correct in some cases and not in others. As an example of this we may present the case when a critical operator receives data stream from two different sources, one is another critical operator and the other is a non-critical operator. If both data streams represent the same data then this construction is valid, but if the data stream of the two operators have different meanings, and the data stream of the non-critical operator was not initialized (using the appropriate command of the CAPS system) then the construction is not valid, and the user will have an error message during the execution of the prototype.

The graph of constraints matrix is an upper triangular matrix. This fact was not utilized during this thesis, but it may help to save storage space and also may introduce some modifications in the algorithms developed in this work in order to speed up the execution time. Our suggestion to the implementors of the algorithm is to spend some time trying to verify the possibilities that this property may produce.

An assumption not explained previously is that any data stream has its associated buffer (responsible for the storage of the data stream). We assumed that the buffer holds only the most recent version of the data stream. This assumption, generally, will not cause problems; but there are some systems that need to have access not only to the most recent data value but also the former data values for analysis of correlation among these streams. One example of system that r.ust to execute the operation described before is an integrated radar network. For this kind of this kind of systems there are two possible solutions that are introduced in the section IV.C.

An important aspect of the two algorithms developed is that they allow interaction with the user without the necessity to rerun all the processes. incremental modifications are possible if the user wants to modify any time constraints of any operator; the only exception is the period. If the user want to modify the period or the precedence constraints then is necessary to rerun the complete algorithm.

One important aspect of incremental modifications is that will be necessary to update the PSDL model of the system under analysis.

B. POSSIBLE CAPS MODIFICATIONS

The analysis in all the previous work done on the development of the CAPS system (and the work presented here) is deterministic in the sense that if a schedule is obtained then it is guaranteed under any load conditions of the system (for the critical operators). This property is obtained at the cost of converting all the sporadic operators to equivalent periodic operators. This kind of conversion imposes a strong restriction on the set of operators, if we have a sporadic operator with a small MCP then it will be necessary to reserve too many slots of time for this operator in the static schedule, even if this operator is hardly triggered.

One possible alternative is to treat all the sporadic operators on a statistical basis, as is done in the dynamic algorithms that work with monotonic rates. The idea is to have a task that will handle all the sporadic operators in a run-time basis. To obtain a reasonable result using this approach we must know in advance the distribution of all the sporadic operators. This special task that handles the sporadic operators will analyze each sporadic operator in the waiting queue and verify which of them is more close to the next statically scheluded critical operator (that is the data flow stream generated by the sporadic operator is an input for the critical operator), then this sporadic operator will be executed during the run time allocated to the special operator. After the execution of this operator the special task will verify if there is another sporadic task that must be executed (searching forward in the statically scheduled operators). This task will be scheduled as the first operator in the static schedule, and will have priority over the non-critical operators when disputing an interval of time. The problems that appear with this idea are that it works on the basis of a ratio of success, as the monotonic rate or the DS algorithm, and it is not clear how to handle the queue of waiting sporadic tasks (which is a FIFO queue by definition in the Ada language).

Other possible modifications to the current CAPS version is the inclusion of *half-critical operators*. This type of operator may represent a non-critical operator that furnishes data for a critical operator. We need to have data available for the critical operator, but we do not have to update this data frequently. The half-critical operator must be handled by both the static scheduler and by the dynamic scheduler. In the static scheduler we ensure that the half-critical operator appears once before the first critical operator that needs its data stream. This will guarantee that at least once in a harmonic length interval of time this operator is executed. After the scheduled time of this half-critical operator by the static schedule the dynamic scheduler will treat this operator as any non-critical, and try to schedule it based on the free slack time and the precedence constraints.

Both modifications suggested above represent an overhead in the Dynamic Scheduler, consequently a careful analysis of the possible improvements obtained versus the lost of velocity of the Dynamic Scheduler must be performed.

The buffers that hold the data streams are one point that may allows some important modifications. The main options are how to handle more than one data element in a stream: queue, or time stamps. All the two possibilities require a definition of the maximum number of data values allowed in each buffer. This number may be defined using the data furnished by the user about period (or minimum calling time) and the finish within time (or maximum response time). The queue option represents sequential analysis of data which probably will not adversely affect the dynamic scheduler. The time stamp option is the one that allows more control over the data being analyzed. This option will allow a search of all available data values and the selection of one based on a time stamp associated with the data (especially in a multiprocessor system when each operator may need to process data from different time intervals), but at the same time will need a more elaborate system to control this search, which will represent an overhead in the system.

C. SOME NEW CONCEPTS

In this thesis we worked only with the case of a single processor, but many of the embedded hard real-time are running in a multiprocessor systems (sometimes spread over long distances and needing a network of telecommunications in order to convey the information). In this kind of scenario a new concept of *contiguity* may be useful. Contiguity is a new kind of constraint over the operators. It requires that some operators must be executed in a same processor (this concept is suggested in [Ref Ste82] for the case of production machines). This new set of constraints may help to simplify the complex problem of scheduling with multiprocessors. Based on the insights obtained through the development of this thesis there are two possible ways to define the set of contiguity constraints: the user may define these constraints; or the system may define them using the job modular decomposition over the set of operators to be scheduled. The job modular decomposition is a good tool for this work because it defines the relationship among all the operators in the set, taking in account the precedence constraints (that may include transmission links constraints), and partitions the operators into modules that shares the same properties in relation to all the other operators in the global set.

V. CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

The goals of this effort were to demonstrate the feasibility of designing an optimal algorithm for hard real-time constraints described in a prototyping language, and to provide guidelines for its implementation. This thesis outlines the tools and procedures that are required. The implementation of the procedures defined in this work may uncover some inconsistencies and difficulties with the design, but the global structure and logic of the algorithms have been established and defined.

The main contributions of this work are the concepts of the graph of constraints, the design of an optimal static scheduling algorithm, and the application of the concept of job modular decomposition to the analysis of hard real-time systems.

Job modular decomposition has not been proven to be an optimal algorithm for our application but it opened new possibilities in the analysis of multiprocessor systems, a field that until now does not have any optimal solution for the case where the number of processors is greater than two.

During the development of this thesis the necessity of analyzing multiprocessors solutions for hard real-time systems that are unfeasible with a single processor became clear.

One important aspect of the single processor is its application in weapons systems, where strong limitations in size, weight, and volume often forbid the use of multiprocessor systems. Following this idea we may conclude that many times even a schedule solution with a maximum tardiness greater than zero may represent an improvement in the performance of a weapons system, as occurs with the proportional navigation in missile system that does not guarantee the interception but improves the chance of small miss distance allowing a greater chance to destroy an enemy aircraft.

Our suggestions for further work in the area of scheduling algorithms are the following:

- Implementation and analysis of performance of the optimal enumeration scheduling algorithm,

- A deep analysis about how to handle the sporadic operators, and verify the advantages of leaving the current deterministic behavior of the static scheduler (using the equivalent periodic operator) in favor of using a statistical behavior using some of the ideas introduced in the chapter IV,

- Modifications in the CAPS system to allow multiple types of buffers, spreading in this way the systems that may be modeled,

- Refining the implicit enumeration technique by the use of the concepts of branch and bound discussed in II.C.3.b, trying to improve the efficiency of the algorithm,

- Start a theoretical analysis to extend the CAPS system for the case of multiprocessors.

As a final remark of this work we emphasize that the rapid prototyping system CAPS is not yet a final product, and that all our comments and suggestions are applicable to the current version.

LIST OF REFERENCES

1.[BFR71] Bratley, P., Florian, M., and Robillard, P.,*"Scheduling with Earliest Start and Due Date Constraints"*, Naval Research Logistic Quartely, 18, 4, December 1971.

2.[BM83] Buer, H., and Mohring, R. H., *"A Fast Algorithm for the Decomposition of Graphs and Posets"*, Mathematics Operations Research, 8, 2, May 1983.

3.[BS74] Baker, K. R., and Su, Z. S., *"Sequencing with Due-Dates and Early Start Times to Minimize Maximum Tardiness"*, Naval Research Logistics Quartely, 21, 1974.

4.[BSR88] Biyabani, S. R., Stankovic, J. A., and Ramamritham, K., *"The Integration of Deadline and Criticalness in Hard Real-Time Scheduling"*, IEEE Transactions on Software Engineering, 1988.

5.[CL88] Chung, J. Y., and Liu, J. W. S., *"Performance Algorithms for Scheduling Periodic Tasks to MInimize Average Error"*, Proccedings of the IEEE 9th Real-Time Symposium, Alabama, December 1988.

6.[Coo71] Cook, S. A., *"The Complexity of Theorem-Proving Procedures"*, Proceedings of the Third Annual ACM Simposium on the Theory of Computing, Saker Heights, Ohio, 1971.

7.[CPS85] Corneil, D. G., Perl, Y., and Stewart, L. K., *"A Linear Recognition Algorithm for Cographs"*, SIAM Journal Comput., 14, 4, November 1985.

8.[CR83] Coolahan, J. E., and Roussopoulos, N., *"Timing Requirements for Time-Driven Systems Using Augmented Petri Nets"*, IEEE Transaction on Software Engineering, 9,5, September 1983.

9.[CS82] Cunningham, W. H., and Siam, L., *"Decomposition of Directed Graphs"*, Alg. Disc. Meth., 3, 2, June 1982.

10.[Cun82] Cunningham, W. H., *"Decomposition of Directed Graphs"*, SIAM Journal Alg. Disc. Meth., 3, 2, June 1982.

11.[GJ78] Garey, M. R., and Johnson, D. S., *"Computers and Intractibility: A Guide to the Theory of N-Completeness"*, W. H. Freeman, San Francisco, 1978.

12.[Gol77a] Golumbic, M. A., *"Comparability Graphs and New MAtroids"*, Journal of Combinatorial Theory (B), 22, 1977.

13.[Gol77b] Golumbic, M. A., *"The Complexity of Comparability Graph Recognition and Coloring"*, Computing, 18, 1977.

14.[Hor74] Horn, W. A., *"Some Simple Scheduling Algorithms"*, Naval Research Logistics Quartely, 21, 1974.

15.[Jan88] Janson, D. M., *"A Static Scheduler for the Computer Aided prototyping System : An Implementation Guide"*, M.S. thesis, Naval Postgraduate School, Monterey, CA, September 1988.

16.[Kic88] Office of Naval Research Kickoff Workshop, *"Foundations of Real-Time Computing Research Initiative"*, Falls Church, Virginia, 1988.

17.[Kil89] Kilic, M., *"Static Schedulers foı Embedded and Hard Real-Time Systems"*, M. S. thesis, Naval Postgraduate School, Monterey, CA, December 1989.

18.[Law78] Lawer, E. L., *"Sequencing Jobs to Minimize Total Weighted Completion Time Subject to Precedence Constraints"*, Ann. Discrete Mathematics, 2, 1978.

19.[LG88] Luqi, and Galik, D., *"An Integrated Tool Environment for Embedded Real-Time Software"*, Technical Report NPS52-88-008, Naval Postgraduate School, Monterey, CA, April 1988.

20.[LK88] Luqi, and Ketabchi, M., *"A Computer Aided Prototyping System"*, IEEE Transactions on Software Engineering, March 1988.

21.[LL72] Liu, C. L., and Laylan, J. W., *"Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment"*, Journal of ACM, 20, 1, January 1972.

22.[LR78] Lenstra, J. K., and Rinnooy, A. H. G. K., *"Complexity of Scheduling under Precedence Constraints"*, Operations Research,26, 1978.

23.[LS87] Leveson, N.G., and Stolzy, J. L., *"Safety Analysis Using Petri Nets"*, IEEE Transactions on Software Engineering, March 1987.

24.[LSS88] Lehoczky, J. P., Sha, L., and Stronsider, J. K., *"Enhanced Aperiodic Responsiveness in Hard Real-Time Environments"*, IEEE Real-time Symposium, 1987.

25.[LTJ85] Locke, C. D., Tokuda, H., and jensen, E. D., *"A Time-Driven Scheduling Model for Real-Time Operating Systems"*, Technical Report, Carnegie-Mellon University, 1985.

26.[Luq88] Luqi, *"Software Evolution Via Prototyping"*, Technical Report NPS52-88-039, Naval Postgraduate School, Monterey, CA, September 1988.

27.[Luq89] Luqi, *"Handling Timing Constraints in Rapid Prototyping"*, IEEE Transactions on Software Engineering, 1989.

29.[LV88] Luqi, and Berzins, V., *"Rapidly Prototyping Real-Time Systems"*, Technical Report NPS52-87-005, Naval Postgraduate School, Monterey, CA, 1987.

30.[Mar88] Marlowe, L., *"A Scheduler for Critical Time Constraints"*, M.S. thesis, Naval Postgraduate School, Monterey, CA, September 1988.

31.[MB83] Mohring, R. H., and Buer, H., *"A Fast Algorithm for the Decomposition of Graphs and Posets"*, Mathematics of Operations Research, 8, 2, May 1983.

32.[MJ89] Muller, J. H., and Spirand, J., *"Incremental Modular Decomposition"*, Journal of ACM, 36, 1, January 1989.

33.[Mok85a] Mok, A., *"A Graph-Based Computation Model for Real-Time Systems"*, IEEE Proceedings of the International Conference on Parallel Processing, Pennsylvania State University, PA, August 1985.

34.[Mok85b] Mok, A., *"Modeling and Scheduling of Dataflow Real-Time Systems"*, IEEE Proceedings of Real-Time Systems Symposium, San Diego, CA, December 1985.

35.[MR84] Mohring, C. L., and Radermacher, F. J.,*"Substitution Decomposition for Discretes Structures and Connections with Combinatorial Optimization"*, Ann. Disc. Math., 19, 1984.

36.[MS79] Monma, C. L., and Sidney, J. B., *"Sequencing with Series-Parallel Precedence Constraints"*, Mathematics Operations Research, 4, 1979.

37.[MS87] Monma, C. L., and Sidney, J. B., *"Optimal Sequencing Via Modular Decomposition: Characterization of Sequencing Functions"*, Mathematics of Operations Research, 12, 1, February 1987.

38.[O'He88] O'Hern, J. T., *"A Conceptual Design for a Static Scheduler for a Hard Real-Time Systems"*, M. S. thesis, Naval Postgraduate School, Monterey, CA, September 1988.

39.[PE72] Pneuli, A., and Even, S., *"Permutation and Transitive Graphs"*, Journal of ACM, 19, 3, July 1972.

41.[PLE71] Pneuli, A., Lempel, A., and Even, S., *"Transitive Orientation of Graphs and Identification of Permutation Graphs"*, Can. Journal of Mathematics, XXIII, 1, 1971.

42.[SF70] Shevrin, L. N., and Fillipov, N. D., *"Partially Ordered Sets and their Comparability Graphs"*, Translation from the Sibirkii Matematicheskii Zhurnal, 11, 3, May-June 1970.

43.[Sha88] Sha, L., *"The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High priority Ada Tasks"*, Technical Report SEI-SSR-4, Software Engineering Institute, March 1988.

44.[Sid81] Sidney, J. B., *"A Decomposition Algorithm for Sequencing with General Precedence Constraints"*, Mathematics of Operations Research, 6, 2, May 1981.

45.[Smi56] Smith, W. E., *"Various Optimizer for Single-Stage Production"*, Naval Research Logistic Quartely, 3, 1956.

46.[Spi85] Spirand, J., *"On Comparability and Permutation Graphs"*, SIAN Journal Computing, 14, 3, August 1985.

47.[SRC87] Stankovic, J. A., Ramamritham, K., and Cheng, S. C., *"Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey"*, COINS Technical Report 87-55, June, 1987.

48.[SS86] Sidney, J. B., and Steiner, G., *"Optimal Sequencing By Modular Decomposition: Polynomial Algorithms"*, Operations Research, 34, 4, July-August 1986.

49.[Ste82] Steiner,G., *"Machine Scheduling with Precedence Constraints"*, Ph.D. dissertation, University of Waterloo, Ontario, Canada, 1982.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
Cameron Station
Alexandria, Virginia 22304-6145

2. Dudley Knox Library 2
Code 0142
Naval Postgraduate School
Montery, CA 93943

3. Center for Naval Analysis 1
4401 Ford Avenue
Alexandria, VA 22303-0268

4. Director of Research Administration 1
Attn: Prof. Howard
Code 012
Naval Postgraduate School
Monterey, CA 93943

5. Chairman, Code 52 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

6. Chief of Naval Research 1
800 N. Quincy Street
Arlington, Virginia 22217

7. National Science Foundation 1
Division of Computer and Computation Research
Attn: Tom Keenan
Washington, D.C. 20550

8. Naval Postgraduate School 50
Code 52Lq
Computer Science Department
Monterey, CA 93943